UNIVERSIDADE ESTADUAL DE CAMPINAS

PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Este exemplar corresponde à redação final da tese defendida por .....
e aprovada pela Comissão Julgadora em ..... / .....

Orientador

"TÉCNICAS DE ESCALONAMENTO PARA SISTEMAS "HARD REAL TIME" - ANÁLISE E IMPLEMENTAÇÃO"

DISSERTAÇÃO SUBMETIDA À UNIVERSIDADE ESTADUAL DE CAMPINAS PARA OBTENÇÃO DO GRAU DE MESTRE EM ENGENHARIA ELÉTRICA

ALUNO: HELIO AZEVEDO

ORIENTADOR: PROF. DR.MAURÍCIO F. MAGALHÃES

CAMPINAS, 15 DE MARÇO DE 1991

Dedico este trabalho
a minha esposa Glaucia e
ao meu filho Felipe.

RESUMO

Inúmeras aplicações em automação industrial possuem severas restrições de tempo que devem ser observadas para o perfeito funcionamento do sistema. Neste trabalho é apresentado um núcleo de tempo real capaz de satisfazer as restrições de tempo de um conjunto de tarefas, com parâmetros bem determinados, através da utilização de técnicas de escalonamento direcionadas para sistemas "HARD REAL TIME". Como resultado foram obtidos três núcleos, sendo cada um implementado com uma estratégia de escalonamento distinta, aderentes à norma MOSI (Microprocessor Operating Systems Interfaces) da IEEE.

ÍNDICE

1. INTRODUÇÃO

A utilização de computadores em aplicações de tempo real é um fato comum atualmente, entretanto, poucos são os ambientes que levam em consideração o fator tempo na descrição das tarefas que compõem o sistema.

Esse fato deve-se principalmente à ausência de técnicas para manipulação das temporizações, desde a fase de especificação até a fase de implementação.

Para permitir o desenvolvimento confiável de Sistemas de Tempo Real (STR) uma nova abordagem é necessária de modo a garantir que as bases do desenvolvimento sejam centradas nos elementos fundamentais dos projetos de tempo real: temporizações e confiabilidade.

A figura 1.1. apresenta os principais tópicos que deverão ser alvo de atenção para, no futuro, serem criadas as bases científicas para o desenvolvimento de sistemas de tempo real.

Tolerância a Falhas	Inteligência Artificial
Escalonadores	Sistemas Operacionais
Linguagens	Arquiteturas
Comunicação	Banco de Dados

Figura 1.1. Principais Áreas de Pesquisa em STR.

Este trabalho centraliza sua atenção nos escalonadores de tempo real, sendo seu objetivo atingir os seguintes resultados:

- apresentar/analisar as principais linhas de pesquisa em escalonadores de tempo real.
- implementar algoritmos de escalonamento para sistemas monoprocessadores, em ambientes com restrições de tempo críticas.

Paralelamente a esses objetivos, outra motivação para o desenvolvimento deste trabalho foi a necessidade existente no Centro Tecnológico para a Informática - Instituto de Automação (CTI/IA) de um núcleo de tempo real capaz de atender aos projetos que exigissem um tratamento especial para o fator tempo.

A tecnologia para o desenvolvimento de núcleos de tempo real encontra-se atualmente sedimentada, sendo que, inúmeras companhias fornecem sistemas que satisfazem uma grande gama de aplicações [SUY75]. Mesmo a nível nacional, inúmeras instituições desenvolveram núcleos de tempo real, estando entre elas o CTI e a UNICAMP [AZE86, COE86].

Posto isto, uma pergunta surge naturalmente: Qual o objetivo no desenvolvimento de mais um núcleo? A resposta a essa pergunta foi parcialmente respondida nos paragráfos anteriores, e representa a motivação de todo este trabalho: tratamento mais eficiente e rigoroso do fator tempo.

Pela sua complexidade, somente esse elemento seria justificativa para tal empreitada, entretanto, os resultados obtidos foram além desse objetivo.

Toda a interface do núcleo com o usuário segue um padrão internacional (norma MOSI [IEE87]) de modo a garantir o transporte da aplicação para outros ambientes.

Conjugado com o núcleo existe um configurador, capaz de modelar o sistema às necessidades de cada usuário, dimensionando-o de acordo com a aplicação.

Sua construção modular, numa linguagem de alto nível, permite fácil adaptação a diferentes microprocessadores; existem atualmente versões para dois micros: MC68000 e 8086/88.

Com o objetivo de criar condições para a análise de técnicas de escalonamento, foram construídos três núcleos distintos, cada um voltado para um algoritmo diferente de escalonamento.

Para apresentar com maior profundidade os resultados obtidos, este trabalho está organizado da seguinte forma.

O capítulo 2 descreve as características dos sistemas de tempo real, além de apresentar as principais linhas de pesquisa a serem adotadas no desenvolvimento desses sistemas.

O capítulo 3 apresenta a evolução das técnicas de escalonamento através de um breve histórico, concluindo com as linhas de pesquisa existentes em escalonamento para tempo real.

O capítulo 4 apresenta os núcleos implementados, suas características, funções, detalhes de seus algoritmos de escalonamento, bem como uma descrição do configurador.

No capítulo 5 o trabalho é encerrado apresentando as conclusões obtidas neste desenvolvimento.

A documentação do sistema é realizada através de três manuais de usuário, sendo um para cada versão, além da documentação de software.

No anexo 1 são apresentados os "tempos e tamanhos" de cada versão obtida.

No anexo 2 são representadas as telas geradas pelo configurador do núcleo durante o diálogo mantido com o usuário.

Para não aumentar consideravelmente o volume deste trabalho, os manuais do usuário não foram incluídos, entretanto, todo o material está à disposição para uma consulta mais ampla [AZE89a, AZE89b].

2. SISTEMAS DE TEMPO REAL

O computador tem se tornado um componente crítico em aplicações que abrangem desde o controle de terminais bancários até manipuladores de robôs ou controle de vôo de aviões.

Para estes sistemas não é suficiente que o software esteja logicamente correto, ou seja, implemente o algoritmo desejado; o sistema deve também atender aos requisitos de tempo dos eventos computacionais, tais como: tempo máximo de resposta a um estímulo externo ou atualizações periódicas de estruturas de dados que representam os eventos sob observação.

Estas restrições de tempo são normalmente determinadas pela aplicação em execução e devem ser respeitadas sob pena de gerar eventos catastróficos [MOK84].

Sistemas de computação que operam sob severas restrições de tempo são denominados "hard real time"; por outro lado, quando as tarefas devem ser executadas rapidamente, mas não possuem tempos críticos para seu término, o sistema é denominado "soft real time".

Neste trabalho estamos particularmente interessados nos sistemas "hard real time". Estes sistemas possuem custo extremamente alto de desenvolvimento sendo a fase de testes realizadas no próprio ambiente ou através de complexos simuladores.

No futuro, a tendência dos sistemas "hard real time" é tornarem- se cada vez maiores e mais complexos. A manipulação de tais sistemas exigirá um esforço conjunto de desenvolvimento em diversas áreas da ciência da computação garantindo as bases científicas necessárias para alcançar os resultados desejados [STA88].

O próximo item apresentará as principais áreas da computação que devem ser abordadas nesse esforço.

2.1 As Bases de Sistemas Hard Real Time

Nas três últimas décadas o avanço realizado nas diversas áreas da ciência da computação tem sido significativo; como exemplo podemos observar a evolução ocorrida na especificação e verificação de sistemas.

Inicialmente as metodologias se fixaram na programação como, por exemplo, os trabalhos de Dijkstra (1966) em programação estruturada; numa segunda fase se concentraram no projeto, sendo resultado significativo as técnicas de projeto estruturado elaboradas por Parnas (1972) e Constantine (1974).

Com as técnicas de programação e projeto sedimentadas os pesquisadores voltaram-se para a análise do problema, um passo significativo nessa direção foi o desenvolvimento da análise estruturada por Ross (1977) e DeMarco (1978).

Na área de sistemas de tempo real as técnicas de Hatley e Ward- Mellor [FAL88a] são atualmente as mais conhecidas; entretanto, apesar desses e de outros trabalhos, os sistemas de tempo real não têm sido contemplados com a devida atenção. Elementos como restrições de tempo, períodos e tolerância a falhas, são considerados somente na fase de implementação e testes do sistema.

A construção de uma ciência que atenda os requisitos de tempo e confiabilidade exige esforços de pesquisa em áreas distintas. Apesar de cada uma dessas áreas conter teorias e técnicas bem desenvolvidas, nenhuma atualmente está direcionada para as questões centrais dos sistemas "hard real time": tratamento coerente do fator tempo, tolerância a falhas e sistemas distribuídos de grande porte.

As principais áreas que serão alvos de atenção na construção de uma metodologia para desenvolvimento de sistemas "hard real time" são descritas a seguir [STA88]:

- *Especificação e verificação.*

A especificação de sistemas é responsável pela decomposição do sistema em partes funcionais e descrição de como esses elementos relacionam-se entre si e com o meio externo.

Qualquer pesquisador que se defrontou com o problema de representar um sistema de tempo real através de técnicas convencionais de análise estruturada sabe das dificuldades encontradas para representar elementos como: interrupções, relógios e restrições de tempo do sistema.

Geralmente, o resultado obtido permite somente uma noção geral do sistema, adiando-se a representação dos elementos críticos até a fase de implementação e testes.

O desafio consiste em introduzir claramente o fator tempo nessa descrição e garantir que a implementação respeite essas restrições.

- *Sistemas operacionais de tempo real.*

O desenvolvimento de sistemas operacionais distribuídos de tempo real apropriados para aplicações complexas como: equipes de robôs trabalhando em conjunto, ou aplicações de comando e controle, necessita que os elementos descritos abaixo sejam dimensionados apropriadamente.

. O fator tempo seja o princípio central do sistema;
. tolerância a falhas;
. alocação de recursos respeitando os "deadlines" e ordem de precedência das tarefas.

- *Linguagens de programação de tempo real.*

Inúmeros modelos de programação para processamento paralelo foram propostos na literatura. Por exemplo, o modelo de tarefas da linguagem ADA, ou o mecanismo de processos distribuídos proposto por Hansen [HAN78]. Esses modelos diferem basicamente nos mecanismos fornecidos para comunicação e sincronização entre processos.

Entretanto, esses modelos não fornecem elementos para explicitamente descrever as restrições de tempo existentes no sistema.

- *Base de dados distribuída para sistemas de tempo real.*

Naturalmente, o acesso a base de dados para tempo real deve ser extremamente rápido. Mas, paralelamente a essa necessidade, outro fator importante é a integração do algoritmo de escalonamento com o algoritmo responsável pelo tratamento de acessos concorrentes à base de dados, de forma a garantir que os requisitos de tempo do sistema sejam respeitados.

- *Inteligência artificial.*

A utilização de IA em algoritmos de escalonamento tem sido descrita em inúmeros trabalhos [ZHA87b, MA82, EFE82, MA84, RAM84, ZHA87a] para resolver problemas de escalonamento NP-hard; entretanto, técnicas específicas para a resolução dos problemas de tempo real ainda necessitam ser aprimoradas.

- *Tolerância a falhas.*

A forte integração entre o sistema de tempo real e a aplicação é responsável pelos rígidos critérios de tolerância a falhas exigidos por esses sistemas.

A ocorrência de falhas em qualquer sistema é um evento inevitável; entretanto, em sistemas de tempo real as consequências podem ser catastróficas como, por exemplo, no controle de usinas nucleares ou no controle de aviões.

Uma vez que falhas são inevitáveis o problema é equacionar seu tratamento de forma a minimizar as consequências sobre o sistema.

Nesse sentido, o desenvolvimento de técnicas para formalmente especificar os requisitos de confiabilidade, levando em consideração as restrições de tempo, é fundamental para o perfeito funcionamento dos sistemas de tempo real.

- *Arquiteturas.*

Muitos dos sistemas de tempo real são distribuídos, sendo cada nó composto por mono ou multiprocessadores.

Nos sistemas de TR convencionais a arquitetura é direcionada para a aplicação. Esse fato leva a um alto custo de desenvolvimento e manutenção, onde pequenas alterações podem exigir o redimensionamento de todo o sistema.

O alto grau de integração obtido nos componentes eletrônicos permite a construção de sistemas que abrangem uma larga faixa de aplicações, desde que organizados de forma apropriada.

Dessa forma, o desenvolvimento de tecnologia em arquiteturas de tempo real, levando em consideração itens como: topologia de interconexão, comunicações, tolerância a falhas, suporte para algoritmos de escalonamento e sistemas operacionais, é uma necessidade importante.

- *Comunicações.*

A necessidade por sistemas distribuídos acentuou um problema latente: comunicação rápida e segura entre os nós da rede levando em consideração as necessidades do algoritmo de escalonamento e do sistema operacional.

- *Escalonadores de tempo real.*

O objetivo do escalonador, em sistemas de tempo real com restrições severas de tempo, é garantir que o maior número possível de tarefas tenha os seus requisitos de tempo cumpridos. Para atingir esse objetivo é necessário equacionar as seguintes váriaveis:

. "deadlines" das tarefas,
. ordem de precedência entre as tarefas,
. alocação de recursos,
. arquitetura do ambiente computacional,
. tipo de tarefas (periódica, aleatória),
. grau de interconexão entre as tarefas.

O equacionamento desse problema frequentemente resulta num problema NP-hard, fato que torna proibitivo sua implementação computacionalmente [FRE86].

Dessa forma, soluções heurísticas são adotadas quando o escalonamento deve ser "on line".

O próximo capítulo deste trabalho discute em detalhes os algoritmos de escalonamento associados às diversas arquiteturas existentes.

3. ALGORITMOS DE ESCALONAMENTO

Entre os recursos oferecidos numa unidade computacional o processador é tradicionalmente o mais concorrido e, como consequência, o mais importante.

A partir do momento em que mais de um usuário tem acesso aos recursos computacionais, critérios devem ser elaborados para determinar qual deverá ser ativado primeiro.

O elemento responsável por essa decisão é denominado escalonador ("scheduler") e o algoritmo utilizado é chamado algoritmo de escalonamento ("scheduler algorithm") [TAN87].

Os critérios adotados no algoritmo de escalonamento estão intimamente associados à utilização destinada ao sistema em questão.

Como forma de avaliar se o escalonador atende aos requisitos do sistema, diversos parâmetros podem ser levantados baseados em dados estatísticos do sistema. Entre os parâmetros mais comuns temos:

- *Utilização do processador.* Nos primeiros sistemas computacionais o alto custo do processador exigia sua utilização massiva de modo a justificar os custos envolvidos.
  Desse modo, o parâmetro "utilização do processador" fornece uma medida da ocupação do sistema e é fornecido através da seguinte relação [HAN73]: tempo de execução / tempo total.

- *"Turnaround time" ou "Response time".* Do ponto de vista do usuário o interesse maior reside no tempo em que o seu "job" está sendo processado, ou seja, quanto mais rápido for o retorno do "job" melhor é o sistema.
  Esse parâmetro é medido através da expressão [MAD74]:tempo_final-tempo_de_chegada.

- *"Throughput".* Para o gerente do sistema computacional uma medida de desempenho fundamental é o número de "jobs" que são executados no sistema.
  Esse parâmetro é fornecido através do número de "jobs" executados por unidade de tempo [HAN73].

O escalonador executa sua função cada vez que a tarefa em execução é interrompida sendo, portanto, um dos mais frequentes programas executados pelo sistema. Desta forma ele deve ser extremamente eficiente para minimizar o "overhead" introduzido pelo sistema operacional.

Este capítulo descreve, inicialmente, os algoritmos clássicos utilizados nos sistemas computacionais para, posteriormente, apresentar algoritmos de escalonamento voltados para aplicações de tempo real.

3.1. Gerência de Processos

A parte do sistema operacional responsável pelo controle das tarefas é denominada gerência de processos.

Basicamente o gerente de processos deve implementar as seguintes funções [MAD74]:

1. Controlar o estado das tarefas.

2. Decidir qual tarefa recebe o processador, quando e por quanto tempo.

3. Alocar processador para as tarefas.

4. Desalocar tarefa dos processadores.

Neste trabalho estamos particularmente interessados na função 2 citada acima, que é realizada pelo algoritmo de escalonamento.

Antes de detalhar os aspectos da gerência de processos é necessário definir alguns termos que serão utilizados neste trabalho.

Computação é um conjunto finito de operações aplicadas a um conjunto finito de dados num esforço para resolver um problema.

Se uma computação resolve um problema ela é chamada algoritmo.

Uma descrição formal de uma computação é chamada programa, e a linguagem na qual ela é expressa é chamada linguagem de programação [HAN73].

Uma tarefa é uma computação que pode ser realizada concorrentemente com outras computações.

Um "job" é uma coleção de atividades necessárias para realizar o trabalho requerido pelo usuário.

Um "job" pode ser dividido em vários passos ("jobs step") executados sequencialmente como, por exemplo: compilação, carga e execução [MAD74];

cada "job" pode gerar várias tarefas.

O objetivo de um algoritmo de escalonamento é fornecer para um dado conjunto de tarefas, uma sequência de execução onde as restrições fornecidas para as tarefas sejam respeitadas, sempre que tal sequência seja viável.

Quando todas as características das tarefas são conhecidas "a priori" o escalonamento é dito ser estático, em caso contrário, dinâmico.

Um algoritmo de escalonamento estático é ótimo se, para qualquer conjunto de tarefas, ele produz um escalonamento que satisfaz as restrições das tarefas sempre que qualquer outro algoritmo possa realizá-lo.

Um algoritmo de escalonamento dinâmico é ótimo se, ele produz um escalonamento factível sempre que um algoritmo de escalonamento estático, com conhecimento completo de todas as tarefas possíveis, possa fazê-lo.

Tradicionalmente, a gerência de processos é dividida em níveis de abstração ("multilevel scheduling"), percorrendo o caminho desde o usuário até o hardware, dividindo o controle entre os níveis existentes, procurando alocar nos níveis inferiores o controle mais simples (logo, mais rápido) e nos superiores o controle mais complexo.

No nível mais baixo temos o "dispatcher" que é ativado quando o processador é liberado para executar uma nova tarefa. Ele retira, da fila de prontos, a próxima tarefa a ser executada.

No nível intermediário temos o "short-term scheduling", cuja função é inserir uma tarefa na fila de prontos segundo a estratégia de escalonamento adotada.

No nível superior temos o "long-term scheduler", o qual realiza ajustes complexos no sistema, que são realizados com uma frequência menor que os anteriores como, por exemplo, a reavaliação das prioridades das tarefas.

Neste trabalho apresentaremos os algoritmos de escalonamento de forma conjunta sem associá-los aos níveis descritos acima.

Essa aproximação foi tomada pelo fato da própria literatura não convergir para um ponto comum a respeito do assunto [MAD74, HAN73, TSI74], além de tal debate não ser objeto deste trabalho.

3.2. Evolução dos Algoritmos de Escalonamento

Os primeiros sistemas computacionais tinham como características principais: o alto custo do processador, monoprogramação e E/S lentas.

Nesses sistemas era comum utilizar um computador satélite responsável pela leitura dos "jobs" e posterior armazenamento em fitas.

O computador principal recebia essa fita como entrada e processava os "jobs" armazenados em sequência, o que caracterizava o sistema "batch" (Figura 3.1.).

Um típico sistema "batch" é o IBSYS para o IBM7090 com um IBM1401 como computador satélite [TSI74].

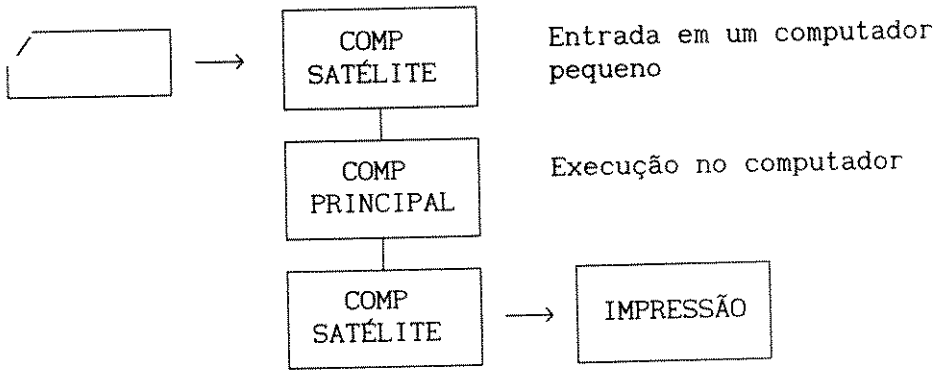

Figura 3.1. Processamento batch.

Em um sistema "batch" o algoritmo de escalonamento é muito simples, consistindo em executar os jobs em sequência a partir da ordem de chegada.

Com o objetivo inicial de permitir interação entre os usuários e os sistemas computacionais e aumentar a eficiência na utilização dos recursos surgiram, a partir da década de 60, sistemas que permitiam o compartilhamento dos recursos por vários usuários.

Esses sistemas são denominados sistemas interativos (Figura 3.2.) implementados através de multiprogramação.

Um dos primeiros S.Os. a utilizar este conceito foi o CTSS desenvolvido no MIT em 1962 [COR62].

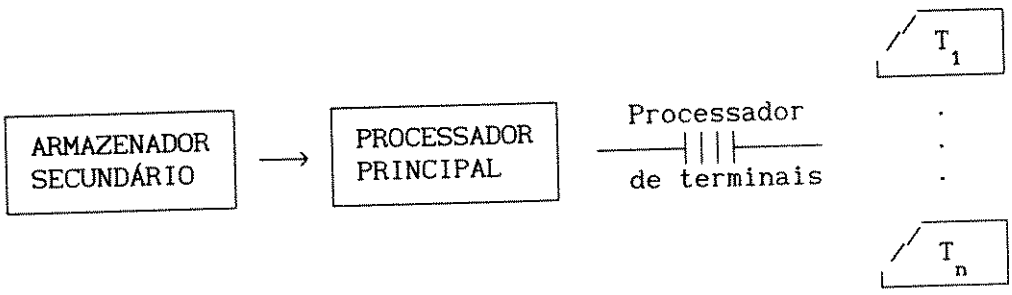

Figura 3.2. Sistema interativo.

Naturalmente os algoritmos de escalonamento *utilizados em sistemas "batch"* não podem ser utilizados em sistemas interativos, pois exigiriam que os usuários aguardassem minutos ou horas antes de obter uma resposta.

Desta forma, técnicas como "time slicing" foram criadas para garantir que um usuário não monopolizasse o processador, alocando para cada "job" um intervalo de tempo denominado "time slice" ou "quantum" para seu processamento.

Ao final de cada "quantum" o sistema elege um novo usuário para utilizar o processador.

Os critérios para a escolha do usuário são obtidos através da análise dos seguintes parâmetros: prioridade, tamanho do "job", taxa de utilização de I/O, tempo de execução, etc.

O aumento da complexidade nas aplicações de tempo real foi fator preponderante no desenvolvimento dos sistemas distribuídos e dos multiprocessadores que, novamente, exigiram alterações nos critérios de escalonamento vigentes.

A utilização de vários processadores acoplados aumentou a complexidade do algoritmo de escalonamento; modelos determinísticos e probabilísticos estão sendo utilizados nos escalonadores com o objetivo de otimizar a distribuição de tarefas entre os processadores existentes no sistema [HWA84].

3.3. "Non-Preemptive X Preemptive"

A estratégia de permitir que uma tarefa execute até o seu término, ou até ela explicitamente liberar o processador, é chamada "non-preemptive".

Os algoritmos de escalonamento pertencentes a este grupo têm como principal característica a simplicidade de implementação, visto que o número de trocas de tarefas na utilização do processador será reduzido.

Em alguns ambientes pode ser necessário interromper uma tarefa temporariamente e alocar o processador para tarefas mais prioritárias; esse tipo de controle é denominado "preemptive".

Na estratégia "preemptive", várias tarefas podem estar no estado de execução em um dado instante, organizadas numa fila denominada pronto.

Neste grupo, cada tarefa recebe o processador por um período limitado de tempo, sendo então removida, cedendo o processador a uma outra tarefa.

Embora a estratégia "preemptive" otimize o tempo de resposta de tarefas pequenas, o custo envolvido aumenta devido ao tempo requerido para a comutação das tarefas.

3.4. Linhas de Pesquisa em Escalonadores

Até o momento, o problema de escalonamento foi apresentado informalmente, ressaltando seus aspectos históricos e sua ligação com a ciência da computação.

Neste item apresentaremos uma descrição um pouco mais profunda a respeito do assunto, além das principais linhas existentes no esforço para resolver o problema.

A terminologia da teoria de escalonamento nasceu nas indústrias de processamento e de manufatura, onde o objetivo é otimizar a produção através do controle de n "jobs" que serão executados por m máquinas.

Entretanto, sua utilização é corrente em áreas que abrangem desde ciência da computação até o controle de tráfego num aeroporto [FRE86].

De forma geral o problema consiste de n "jobs" $\{J_1, J_2, \ldots, J_n\}$ que serão executados por m máquinas $\{M_1, M_2, \ldots, M_m\}$.

Cada "job" passa através de uma máquina uma única vez, sendo seu processamento denominado "operação".

As restrições inseridas sobre o problema são denominadas "restrições tecnológicas" e determinam uma ordem de processamento para os "jobs".

Cada "operação" requer uma certa quantidade de tempo para ser concluida; esse tempo é denominado "tempo de processamento" e é representado por P_{ij}.

O tempo R_i denominado "tempo de pronto" é o tempo no qual o "job" J_i está disponível para execução.

O problema é encontrar uma sequência, na qual os "jobs" passem pelas máquinas, que:

a) seja compatível com as restrições tecnológicas,

b) ótima com respeito a algum critério de desempenho.

Este não é um problema fácil de resolver; em princípio existem $(n!)^m$ soluções, corretas ou incorretas, para o problema [FRE86].

Suponha por exemplo 5 "jobs" e 4 máquinas, o que resulta em 2.1×10^8 possibilidades. Em um computador capaz de analisar 1000 escalonamentos por segundo, o problema seria resolvido em cerca de 58 horas.

Os algoritmos para problemas de escalonamento podem ser divididos em duas classes: algoritmos construtivos e algoritmos de enumeração.

Os algoritmos construtivos são algoritmos que obtêm uma solução ótima dos dados do problema segundo um conjunto simples de regras que determinam exatamente a ordem de processamento.

Os algoritmos de enumeração listam todos os escalonamentos possíveis e então eliminam os escalonamentos não ótimos da lista. Naturalmente, os escalonamentos não são listados explicitamente, sendo explorados somente aqueles que através de algum método de análise se apresentam mais promissores. Essa técnica é denominada enumeração implícita.

Como era de se esperar, os algoritmos construtivos fornecem a solução mais rapidamente do que os de enumeração; o problema é que nem sempre é possível obtê-los, sendo sua existência comprovada somente para alguns casos específicos.

Nos próximos itens são apresentadas as técnicas mais conhecidas de enumeração.

3.4.1. Programação Dinâmica

Nesta classe o problema é decomposto numa sequência de problemas aninhados, sendo a solução de um sub-problema derivada do precedente.

A desvantagem desse método reside na necessidade de manter os dados de todos sub-problemas armazenados para, no final da expansão, obter o escalonamento ótimo.

3.4.2. Método "Branch and Bound"

A representação do problema de escalonamento pode ser realizada através de uma árvore, onde a raiz representa as posições a serem determinadas e as folhas são o preenchimento dessas posições.

Com o auxílio da árvore podemos observar as vantagens da programação dinâmica sobre a enumeração completa. Esta última explora todos os caminhos possíveis da raiz até a extremidade, enquanto a programação dinâmica elimina muitos caminhos possíveis na expansão da árvore.

Como afirmado acima, a desvantagem da programação dinâmica reside na quantidade de recursos para a sua implementação, pois ele se move simultaneamente desde a raiz em todas as direções.

A técnica "Branch and Bound" tenta evitar essa falha explorando a árvore de forma desbalanceada, explorando completamente um ramo antes do outro, além de constantemente verificar se alguma informação se tornou redundante, eliminando-a em caso afirmativo.

Basicamente, o método "Branch and Bound" adota uma função denominada "lower bound" $lb(A)$ que representa o potencial de uma sequência já construída.

Através da análise da função $lb(A)$ é realizada a opção pelo ramo da árvore a ser expandido, sem manter informação dos ramos já percorridos.

3.4.3. Programação Inteira

Muitos pesquisadores adotaram como caminho para a solução do problema de escalonamento sua transformação em programas matemáticos, particularmente programas inteiros.

Apesar de se apresentar como um caminho promissor nem sempre o resultado é o desejado. Os algoritmos de programação matemática são aplicáveis somente em problemas de dimensões reduzidas; os problemas de escalonamento, no entanto, podem ser de dimensão bem maior.

Dessa forma, essas transformações são perigosas pelo fato de que os problemas em programação inteira não são mais fáceis de resolver do que os problemas originais de escalonamento [FREE86].

Entretanto, um breve tratamento será oferecido ao método, somente para completeza do trabalho.

O corpo da teoria e algoritmos chamados de programação inteira se direcionam para problemas do tipo:

$$\text{Minimizar } f(x_1, x_2, \ldots, x_l)$$

com respeito a $x_1, x_2, \ldots, x_l$ sujeitas às restrições:

$$g1(x_1, x_2, \ldots, x_l) \leq b_1$$
$$g2(x_1, x_2, \ldots, x_l) \leq b_2$$
$$g3(x_1, x_2, \ldots, x_l) \leq b_3$$
$$\vdots$$
$$gk(x_1, x_2, \ldots, x_l) \leq bk$$

Ou, em outras palavras, programação matemática é uma família de técnicas para otimizar uma função sujeita a restrições sobre as variáveis independentes.

Em problemas de escalonamento,nós desejamos otimizar uma medida de desempenho sujeita a restrições tecnológicas sobre a ordem de processamento factível.

Dessa forma, é possível transformar o último no primeiro, pois ambos estão relacionados com o problema de otimização com restrições.

Em programação inteira algumas das variáveis independentes são limitadas a assumir valores inteiros; frequentemente os únicos valores permitidos são 0 e 1, indicando a ausência ou presença de uma certa propriedade; além disso, as funções f, g_1, g_2, g_3, ..., g_k são lineares.

Os métodos para resolver esse tipo de problema são baseados em propriedades de programação inteira em geral e não levam em consideração propriedades particulares do problema sendo resolvido.

Como resultado eles tomam mais tempo para obter a solução do que algoritmos de enumeração implícita desenvolvidos especialmente para resolvê-los.

3.4.4. Métodos Heurísticos

As técnicas de enumeração implícita nem sempre podem ser aplicadas na prática pelo grande volume de recursos/tempo necessários para sua implementação.

Dessa forma, se não for possível encontrar um escalonamento ótimo para um problema dentro de um tempo razoável, devemos utilizar nosso conhecimento e experiência para obter um escalonamento que, se não ótimo, pode no mínimo ser melhor que a média. Esse tipo de algoritmo é chamado heurístico.

Assim, as técnicas heurísticas devem ser utilizadas quando: uma solução construtiva em tempo polinomial não existe e as técnicas de enumeração implícita não são computacionalmente aplicáveis.

É interessante observar que as técnicas de enumeração implícita são utilizadas como base em alguns métodos heurísticos; a diferença surge na seleção do elemento a ser expandido.

Como exemplo, podemos utilizar a técnica "Branch and Bound" alterando a função lb(A) de forma a selecionar o "job" com o menor tempo de processamento (SPT - "Shortest Processing Time") [FREE86].

Inúmeros trabalhos têm sido recentemente publicados na área [ZHA87b, MA82, EFE82, MA84, RAM84, ZHA87a], indicando um grande esforço nessa direção.

3.5. Algoritmos Clássicos de Escalonamento

3.5.1. Escalonamento por Ordem de Chegada

Neste escalonamento (FCFS - First Come, First Served) as tarefas são executadas de acordo com a sequência de chegada.

Este procedimento favorece tarefas longas, pois elas deverão monopolizar o processador com prejuízo das tarefas menores.

Este tipo de escalonamento é "non-preemptive".

3.5.2. Tarefa com Menor Tempo de Execução Primeira

Neste escalonamento (Shortest Job Next) uma distribuição mais justa do processador é efetuada entre os usuários, levando em consideração o tempo de execução estimado para cada tarefa, diminuindo o "turnaround" do sistema.

3.5.3. Escalonamento "Round Robin" (RR)

Um dos mais tradicionais métodos de escalonamento preemptivo é o "round robin".

Neste escalonamento cada tarefa recebe um intervalo de tempo, denominado "quantum", durante o qual ela tem o direito de utilizar o processador. Ao final de seu "quantum" a tarefa é retirada de execução dando lugar a outra tarefa.

Com esse esquema, naturalmente existirá uma fila com as tarefas prontas para execução que terão direito ao processador por q segundos ("quantum") (Figura 3.3.).

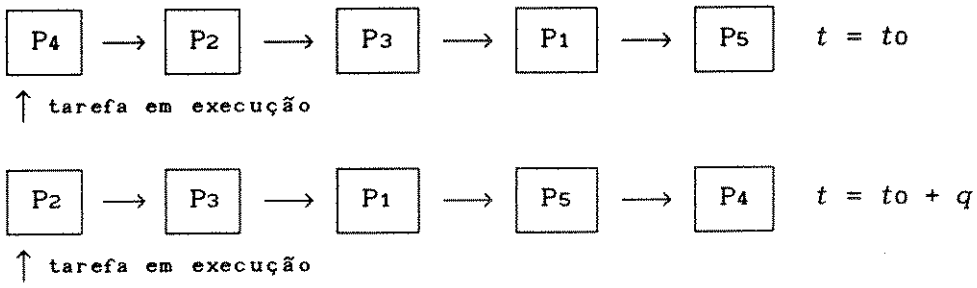

Figura 3.3. Escalonamento RR.

Dessa forma a tarefa no topo da fila de prontos recebe um "time slice" de q segundos, sendo então enviada, após execução, para o fim da fila.

Se existirem K tarefas na fila de prontos então cada tarefa recebe q segundos de todo Kq segundos do tempo do processador (sem levar em consideração o tempo de comutação).

Consequentemente, cada tarefa tem a "ilusão" de estar sendo executada à velocidade de $1/K$ do relógio que sincroniza o processador.

Um ponto importante no escalonamento RR é o tamanho do tempo q.

Se q for infinito teremos o escalonamento por ordem de chegada, se q for muito pequeno o processador utilizará a maior parte de seu tempo controlando as comutações das tarefas.

Portanto, o tempo q deve ser escolhido como um valor intermediário tal que o "overhead" introduzido pelo tempo de comutação seja tolerável.

O modelo básico de escalonamento RR tem sofrido alterações para atender de forma mais eficiente aplicações específicas. Dentre as variações existentes podemos citar:

- *Cycle_Oriented RR.* Neste modelo o tempo de resposta é garantidamente fixado dentro dos limites estabelecidos.

  No ínicio de um ciclo o tempo de resposta mínimo desejado é dividido pelo número (K) de tarefas na fila obtendo o "quantum" do ciclo.

Chegadas durante um ciclo são mantidas numa fila auxiliar para, posteriormente serem adicionadas à fila de prontos no início do próximo ciclo.

- *Biased RR.* O "quantum" recebido pela tarefa não é uniforme, mas sim dependente de outros fatores como: prioridade, utilização de I/O, tamanho da tarefa, tempo de execução, etc.

O escalonamento RR tem sido aplicado principalmente em sistemas interativos, de forma a obter tempos de resposta toleráveis.

3.5.4. Escalonamento por Prioridade

Outra técnica para escalonar a fila de prontos é a seleção da tarefa com a prioridade mais alta.

No modelo "non-preemptive" a tarefa com prioridade mais alta executa até o seu final sem ser bloqueada. Caso outra tarefa seja inserida na fila com prioridade superior à corrente, ela deve aguardar até a tarefa em execução liberar o processador.

No modelo "preemptive", quando surge uma tarefa com prioridade superior à tarefa em execução, esta é interrompida para permitir que a tarefa com prioridade maior entre em execução.

Um ponto básico no escalonamento por prioridade é definir qual parâmetro será levado em consideração para obter a prioridade da tarefa em questão.

Diversas linhas têm sido adotadas como, por exemplo:

. prioridade fornecida pelo usuário

. tarefa com menor tempo de execução

. tarefas com I/O intensivo

Além dessas linhas podemos ainda alterar o valor da prioridade dinamicamente, de acordo com a evolução da tarefa e critérios próprios do Sistema Operacional.

Levando em consideração esses parâmetros, diversas estratégias de implementação têm sido adotadas nos algoritmos de escalonamento.

Dentre as mais importantes podemos citar:

- *I/O bound.* Nesta classe o objetivo é manter as unidades periféricas ocupadas.

  Desta forma, a prioridade das tarefas varia dinamicamente de acordo com a intensidade de utilização das unidades de I/O.

  Especificamente, uma tarefa pode ter a prioridade inversamente proporcional ao período de tempo decorrido desde a última operação de I/O.

- *Linearly Increasing Priority* [KLE70]. Cada tarefa recebe uma prioridade ao entrar no sistema.

  Essa prioridade cresce , a uma taxa "*a*" enquanto está esperando na fila de prontos, e a uma taxa "*b*" enquanto está em execução.

  Diversos mecanismos de escalonamento podem ser obtidos escolhendo apropriadamente os valores "*a*" e "*b*".

  Se tivermos $0 < a \leq b$ e prioridade inicial igual a zero, então a regra de escalonamento será por ordem de chegada.

  Mantendo $a>0$ e $b=0$ e prioridade inicial diferente de zero, teremos a garantia de que com o decorrer do tempo, mesmo as tarefas menos prioritárias ganharão acesso ao processador.

3.5.5. "Feedback Queues"

Nesta estratégia são utilizadas "*n*" filas, cada nível sendo tratado separadamente através de escalonamento realizado pela ordem de chegada (FCFS).

Ao ser criada, a tarefa é inserida no fim da primeira fila, assim que ela receber um "quantum", ou seja, terminar a execução correspondente à fila em questão, ela será inserida no final da fila imediatamente superior à fila anterior (figura 3.4.).

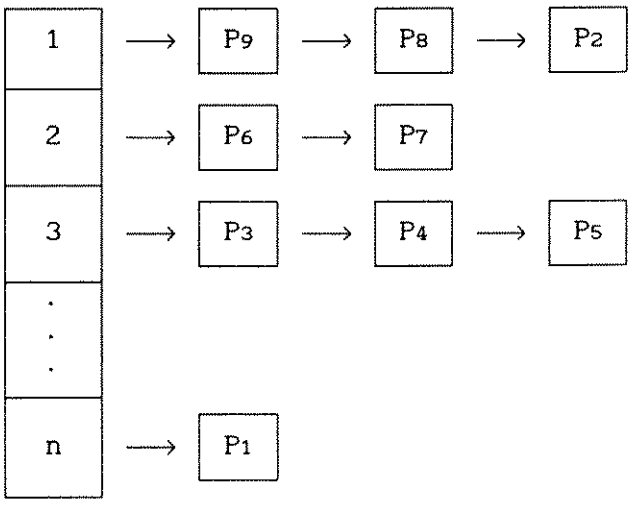

Figura 3.4. Feedback queues.

O algoritmo de escalonamento ativa primeiro as tarefas presentes na fila 1, passando a atender as próximas filas sempre que não existirem mais tarefas na fila atual. Teoricamente o número de filas é infinito; entretanto, na prática isso não é possível, e quando uma tarefa atinge o último nível permanece nesse nível até seu final.

Dessa forma, uma tarefa recém-chegada recebe implicitamente uma prioridade alta e executa pelo menos o mesmo número de "quanta" recebidos pela tarefa menos executada no sistema.

"Feedback Queues" atende preferencialmente às tarefas com tempo de

processamento pequeno obtendo resultados semelhantes à técnica "tarefa com menor tempo de execução primeira", com a vantagem de não possuir qualquer informação a respeito dos tempos de execução das tarefas .
Uma variação desta estratégia é permitir que a tarefa permaneça numa fila por mais que um "quantum" utilizando a estratégia "Round Robin" para gerenciar as tarefas existentes numa fila; a vantagem consiste num menor número de filas para gerenciar.

Outra variação consiste em classificar as tarefas prontas para executar em classes. Tarefas na classe mais alta executam por um "quantum", tarefas na próxima classe executam por dois "quanta" e assim sucessivamente aumentando o tempo de execução por uma potência de dois.

Sempre que uma tarefa utiliza todo o tempo alocado para ela, é rebaixada para a próxima classe.

3.5.6. Escalonamento em Dois Níveis

A maioria dos sistemas operacionais, atualmente, não conserva todas as tarefas prontas para executar em mémoria, mas sim, divide-as entre memória principal e secundária.

Essa divisão se reflete no algoritmo de escalonamento, desde que o tempo de carga de uma tarefa do armazenamento secundário deve ser considerado nas ativações realizadas pelo escalonador.

Uma técnica para gerenciar essa situação consiste em dividir o algoritmo de escalonamento em dois níveis.

O escalonador de nível mais baixo é responsável pela escolha de uma tarefa que esteja na memória, enquanto que o escalonador de alto nível trata da escolha das tarefas que se movimentarão entre a memória principal e a secundária ou vice-versa.

O tipo de escalonamento utilizado em cada um dos escalonadores pode ser qualquer um dos apresentados nos itens anteriores.

3.6. Algoritmos de Escalonamento para Sistemas de Tempo Real

A utilização de computadores em sistemas de tempo real não é recente, na década de 60 já existiam sistemas computacionais atuando no controle de plantas industriais [HEX88].

Entretanto, seu reconhecimento como uma disciplina da ciência da computação só ocorreu no ínicio da década de 80, onde o desenvolvimento de metodologias de software, associadas a novas linguagens e Sistemas Operacionais, possibilitou uma nova postura dos analistas em relação ao problema.

Há alguns anos, métodos de diagramação como os de Yourdon/DeMarco foram estendidos para expressar sistemas de tempo real. Atualmente, técnicas específicas existem para a área.

Os mais conhecidos métodos de análise de sistemas de tempo real, utilizadas em "CASE tools", são os métodos de Hatley [HOW88] e Ward-Mellor [HOW88, WAR85].

Paralelamente a esses desenvolvimentos, o avanço obtido na velocidade de processamento através do aprimoramento dos multicomputadores [HEX88] exigiu a criação de novas técnicas para gerenciar a execução das tarefas de forma coerente pelos diversos processadores existentes no sistema.

Naturalmente, os parâmetros utilizados nos sistemas computacionais tradicionais não servem, a priori, como medida de desempenho de um escalonador para Tempo Real.

Não existe sentido, por exemplo, em exigir que a ocupação do processador e das linhas de I/O sejam máximas, ou otimizar o "throughput", em detrimento da operação correta de uma planta industrial.

Para atender a estes sistemas o problema de escalonamento pode ser descrito pelos seguintes elementos: um conjunto finito de processadores $P = \{P_1, P_2, \ldots, P_r\}$, um conjunto finito de tarefas $T = \{T_1, T_2, \ldots, T_n\}$, um conjunto finito de recursos $R = \{R_1, R_2, \ldots, R_r\}$, e uma relação de precedência entre as tarefas.

O objetivo é obter um algoritmo ótimo, onde todas as tarefas obedeçam as restrições de tempo, utilizando de forma balanceada os recursos disponíveis.

Apesar do enunciado simples, este problema é reconhecido como sendo NP-hard [GAR79], ou seja, não é possível obter um algoritmo ótimo que solucione o problema computacionalmente.

A partir desse fato diversos pesquisadores têm procurado restringir o problema limitando as características do processador [RAY87], das tarefas [ZHA87b], dos recursos [GAR75a, GAR75b] ou das relações de precedência entre as tarefas [MUN70].

3.6.1. Critérios de Classificação

A classificação dos algoritmos de escalonamento para multicomputadores está associada aos elementos descritos a seguir.

a) *Arquitetura do multicomputador.* A forma como os processadores estão interligados afeta o algoritmo através dos custos de comunicação entre as tarefas.

   Assim, temos os multiprocessadores como apresentando os custos mais baixos de comunicação, pois a troca de informações é realizada através de memória comum. Como exemplo de escalonador para multiprocessadores, podemos citar o trabalho de Lai e Sahni [LAI84] que apresenta um algoritmo para escalonar tarefas em *n* processadores com memórias de tamanho variável.

   No outro extremo, os sistemas distribuídos apresentam alto custo, pois as mensagens devem ser enviadas através de complexas redes de comunicação.

b) *Restrições sobre as tarefas.* Como forma de facilitar o controle do escalonador, restrições são impostas sobre as tarefas.

   As mais comuns são associadas ao grau de interação entre as tarefas.

Como exemplos podemos ter tarefas independentes (jobs), ou tarefas que se relacionam através de precedência na ordem de execução ou através de comunicações.

De forma genérica, restrições de precedência limitam a escolha realizada pelo escalonador, exigindo que uma operação dentro de uma tarefa tenha se encerrado antes que uma operação particular dentro de outra tarefa se inicie.

Normalmente, essas relações são representadas através de grafos dirigidos, onde os nós representam as tarefas e os arcos representam precedência entre as tarefas (figura 3.5.).

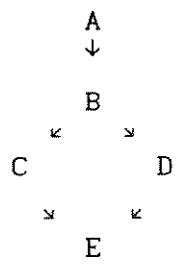

Figura 3.5. Grafo de tarefas.

Outra restrição está associada ao comportamento das tarefas em relação ao escalonador; Stankovic [ZHA87b] por exemplo, divide as tarefas em periódicas e não periódicas.

Finalmente, temos os escalonadores direcionados para aplicações específicas, onde podemos citar o trabalho de Sadayappan e Ercal [SAD87] que propõe um escalonador para programas de modelamento de análise finita.

Outros elementos podem ainda ser levantados, como associação de tarefas com um determinado processador; entretanto, esses elementos não são críticos para a classificação proposta.

c) *Tipos de ativações.* Se todos os dados referentes à ativação das tarefas estão presentes antes do ínicio da execução e o escalonamento é realizado "off_line", a estratégia de escalonamento adotada é dita ser estática.

No extremo oposto, quando variações ocorrem durante a execução do sistema e o escalonamento deve ser processado em tempo de execução, a estratégia é denominada dinâmica.

Se a estrutura das interações do programa paralelo é caracterizada estaticamente durante a compilação, então o escalonamento estático é atrativo pois a análise necessita ser feita somente uma vez e o "overhead" introduzido pelo algoritmo durante a execução é minimizado.

d) *Grau de intervenção.* Outra forma de classificar os algoritmos de escalonamento consiste em dividi-los em preemptivos e não preemptivos.

Nos preemptivos a tarefa pode ser interrompida no meio de sua execução, cedendo o processador a outra tarefa.

No extremo oposto estão os não preemptivos, onde as tarefas mantêm o controle total do processador.

e) *Recursos.* Toda tarefa necessita de recursos para a sua execução, entre os recursos mais comuns temos: memória, I/O, processadores específicos (ponto flutuante, gráficos, etc).

A unidade de processamento também é um recurso; entretanto, devido a sua importância no sistema ela é tratada separadamente.

Podemos dividir os algoritmos de escalonamento, com relação aos recursos, da seguinte forma: com gerência, sem gerência, e gerência parcial.

No primeiro caso, todos os recursos disponíveis no sistema são levados em consideração no algoritmo de escalonamento [ZHA87b].

No outro extremo temos escalonadores sem gerência de recursos, onde esse elemento é ignorado [RAM84], pois é assumido que sejam ilimitados.

Num ponto intermediário, ocorre a gerência de apenas um recurso como, por exemplo, memória.

Como forma de concatenar os elementos de classificação, eles são apresentados na tabela I.

Tabela I. Tabela de Classificação.

Grau de intervenção	tipos de tarefas	Arquitetura	Ativações	Recurso
Preemptivo	jobs	multiproc.	estática	com gerência
	tarefas c/ precedência			
	tarefas c/ comunicação			gerência parcial
não Preemptivo	tarefas p/ aplicações específicas	distribuída	dinâmica	sem gerência

3.6.2. Classificação pelas Ativações

Neste item apresentaremos uma classificação para algoritmos de escalonamento para tempo real adotando uma das linhas de classificação proposta no ítem anterior, que divide os algoritmos em dois grupos: estáticos e dinâmicos.

Existem dois tipos de sistemas de tempo real [CHE87]: "soft real time" e "hard real time".

Em sistemas "soft real time" as tarefas devem ser executadas no menor tempo possível, mas não possuem tempos críticos para atingir o seu final.

Em sistemas "hard real time", as tarefas possuem um rígido requisito sobre os seus tempos de execução ("deadlines"), que devem ser cumpridos sob pena do sistema não operar corretamente.

Neste trabalho estamos particularmente interessados em sistemas "hard real time".

O enunciado geral do problema de escalonamento pode agora ser expandido para representar as restrições de tempo exigidas pelos sistemas "hard real time".

Os seguintes parâmetros representam as restrições:

- O tempo de chegada, A: o tempo no qual uma tarefa se faz presente no sistema;
- o tempo de pronto, R: o primeiro instante no qual a tarefa pode iniciar a execução ($A \leq R$);
- o tempo máximo de computação, C: o tempo de execução é sempre menor ou igual a C;
- o "deadline", D: o tempo no qual a execução da tarefa deve ter sido encerrada.

Possuindo esses parâmetros, a função do escalonador é determinar se existe uma sequência para executar as tarefas tal que as restrições de tempo, recursos e precedência sejam cumpridos.

3.6.2.1. Escalonadores Estáticos

Como já afirmado, os escalonadores estáticos elaboram o escalonamento antes das tarefas entrarem em execução.

Para que o algoritmo de escalonamento possa ser gerado, todas as tarefas, bem como todos os seus parâmetros, devem estar presentes e servir como entrada para o escalonador.

Podemos dividir os escalonadores estáticos em Centralizados (monoprocessadores e multiprocessadores) e Distribuídos.

a) *Sistemas centralizados.* O fato de sistemas centralizados terem surgido antes dos distribuídos justifica o grande volume de trabalhos existentes na área.

   Esses trabalhos podem ainda ser divididos em dois grupos: tarefas independentes e tarefas com restrições de precedência.

   - *Tarefas independentes.* Liu e Layland [LIU73] desenvolveram uma técnica para escalonamento de tarefas periódicas denominada escalonamento "rate-monotonic", cuja idéia central é atribuir prioridades para as tarefas de acordo com seu período. Eles provaram que essa estratégia é ótima para prioridades fixas com uma utilização do processador de até 0.69.

     Este trabalho tornou-se um "clássico" na área de escalonamento sendo posteriormente estendido para atender tarefas aperiódicas [SPR89].

     Horn [HOR74] apresentou um algoritmo para sistemas preemptivos, com uniprocessador, para escalonar tarefas baseado na técnica "earliest deadline first", que foi posteriormente estendido para multiprocessadores.

     O problema se agrava quando a característica não preemptiva é exigida, onde muitas situações são "NP_hard", principalmente quando envolvem multiprocessadores.

Moore [MO069] provou que o algoritmo "earliest deadline first" é ótimo para escalonar um conjunto de tarefas não preemptivas com o mesmo tempo R (tempo de pronto) em monoprocessadores.

Para sistemas multiprocessadores e escalonamento não preemptivo, um algoritmo com tempo polinomial é disponível somente quando o tempo de computação de cada tarefa é unitário [CHE87].

- *Tarefas com restrição de precedência.* Para sistemas com uniprocessador, muitos problemas de escalonamento podem ser resolvidos em tempos polinomiais; Lawler [LAW73] apresenta um escalonador não preemptivo que trata tarefas com "deadlines" e restrições de precedência.

  Em sistemas multiprocessadores o escalonamento com precedência é mais complicado que em unipocessador, como exemplo, escalonar tarefas com precedência arbitrária e tempo de computação unitário é um problema "NP_hard", tanto para sistemas preemptivos como para não preemptivos.

  Dessa forma diversas pesquisas têm procurado obter heurísticas para obter a solução sub_ótima do problema.

  Elsayed [ELS82] apresentou inúmeros algoritmos heurísticos para encontrar soluções subótimas para problemas de escalonamento estático.

b) *Sistemas distribuídos.* Esta categoria tem sido tradicionalmente formulada como um problema de alocação de tarefas.

Alocação de tarefas é um problema complexo em sistemas distribuídos, ainda que sem restrições de tempo, devido aos atrasos gerados pela comunicação entre os processadores.

Como exemplo, podemos citar que o problema de obter uma alocação ótima com um grafo de comunicação arbitrário para quatro processadores com velocidades diferentes é um problema "NP_hard".

Dessa forma, considerável esforço tem sido gasto sobre problemas mais restritos, ou no desenvolvimento de heurísticas para encontrar soluções sub_ótimas.
Um trabalho interessante é o de Ma et al [MA82], que descreveu um

modelo baseado em programação inteira e aplicou o algoritmo heurístico "branch and bound" para resolver o problema de alocação.

3.6.2.2. Algoritmos de Escalonamento Dinâmico

Os algoritmos dinâmicos têm como principal característica o escalonamento em tempo de execução permitindo a ativação de tarefas dinamicamente.

Como os algoritmos dinâmicos são executados conjuntamente com as tarefas, além de sua complexidade natural eles também devem ser extremamente rápidos, caso contrário podem inviabilizar o sistema.

Como no escalonamento estático, o escalonamento dinâmico será dividido em centralizado e distribuído.

a) *Sistemas centralizados.* Segundo o trabalho de Cheng et al [CHE87], muitos dos algoritmos utilizados em escalonadores estáticos centralizados, ao menos em teoria, poderiam ser aplicados dinamicamente; entretanto, somente para alguns poucos casos os algoritmos continuariam sendo ótimos.

   Mok e Dertouzos[MOK78] provaram que, para sistemas com multiprocessadores, não existe algoritmo ótimo para escalonar tarefas com preempção se o tempo de chegada (A) não é conhecido a priori.

   Dessa forma, as técnicas heurísticas, que obtêm soluções sub_ótimas são muito interessantes em sistemas dinâmicos.

   Para sistemas com uniprocessador, Dertouzos provou que ambas as estratégias, "earliest deadline algorithm" e "least laxity scheduling", são ótimas para escalonar tarefas independentes, com preempção e tempos arbitrários de chegada.

   Locke, Tokuda e Jensen [LOC75], realizaram um estudo sobre políticas de escalonamento dinâmico em sistemas centralizados, e concluiram que as políticas "least laxity first" e "earliest deadline first" são as políticas heurísticas que apresentam melhores resultados.

b) *Sistemas distribuídos.* O problema de alocação em sistemas distribuídos é muito mais complexo do que em sistemas centralizados, devido à interação dinâmica que deve existir entre os nós da rede, de forma a obter uma ocupação balanceada do sistema.

Normalmente a aproximação tomada para resolver este problema consiste em dividir o algoritmo de escalonamento em dois níveis: um local e outro distribuído.

No nível local é decidido se a chegada de uma nova tarefa impede ou não a conclusão satisfatória das demais.

No nível distribuído é realizada a análise para verificar qual nó possui melhores condições de receber uma nova tarefa.

Com essa nova estratégia muitos dos algoritmos dinâmicos desenvolvidos para sistemas centralizados podem ser utilizados ao nível local dos algoritmos de escalonamento distribuído.

Quanto ao nível distribuído, novas técnicas foram desenvolvidas e são conhecidas por "load balancing", pois devem balancear a distribuição de tarefas entre os nós da rede.

A análise dos algoritmos "load balancing" pode ser dividida em dois grupos: tarefas independentes e tarefas com restrição de precedência.

- *Tarefas independentes.* A alocação de tarefas independentes foi a primeira a ser abordada pelos pesquisadores. Um trabalho interessante na área é o de Ramamritham e Stankovic [RAM84] que apresenta dois algoritmos responsáveis pela seleção do nó que deverá receber a tarefa para execução.

  O primeiro denominado "focussed addressing", realiza uma estimação a respeito do nó que possue mais chance de executar a nova tarefa. A decisão tomada não é precisa, desde que as informações não são atualizadas; entretanto, ela é rápida.

  O segundo, denominado "bidding", consiste em realizar pedidos e analisar respostas dos nós envolvidos, para eleger o nó apropriado.

Naturalmente os custos envolvidos são superiores ao algoritmo anterior devido às comunicações necessárias.

- *Tarefas com restrição de precedência.* Naturalmente a precedência entre as tarefas é um fator que torna os algoritmos de escalonamento altamente complexos.

  Os algoritmos desenvolvidos para sistemas estáticos e tarefas com restrição de precedência raramente podem ser utilizados em sistemas dinâmicos, pelo fato de exigirem todo o grafo de precedência como entrada.

  Trabalhos de Cheng, Stankovic e Ramamritham [CHE86], abordam o problema dividindo as tarefas em grupos, e associando "deadlines" aos grupos de tarefas.

  Cada grupo é então dividido em subgrupos e alocado utilizando algoritmos descritos para tarefas independentes.

Os algoritmos estáticos possuem uma faixa muito estreita de aplicações pela exigência de que todas as informações estejam presentes antes da execução.

Dessa forma, o interesse maior se concentra nos algoritmos dinâmicos, pelo fato de serem mais apropriados ao tratamento de eventos não periódicos, comuns em sistemas de tempo real.

A figura 3.6. apresenta de forma resumida a classificação proposta para sistemas de tempo real.

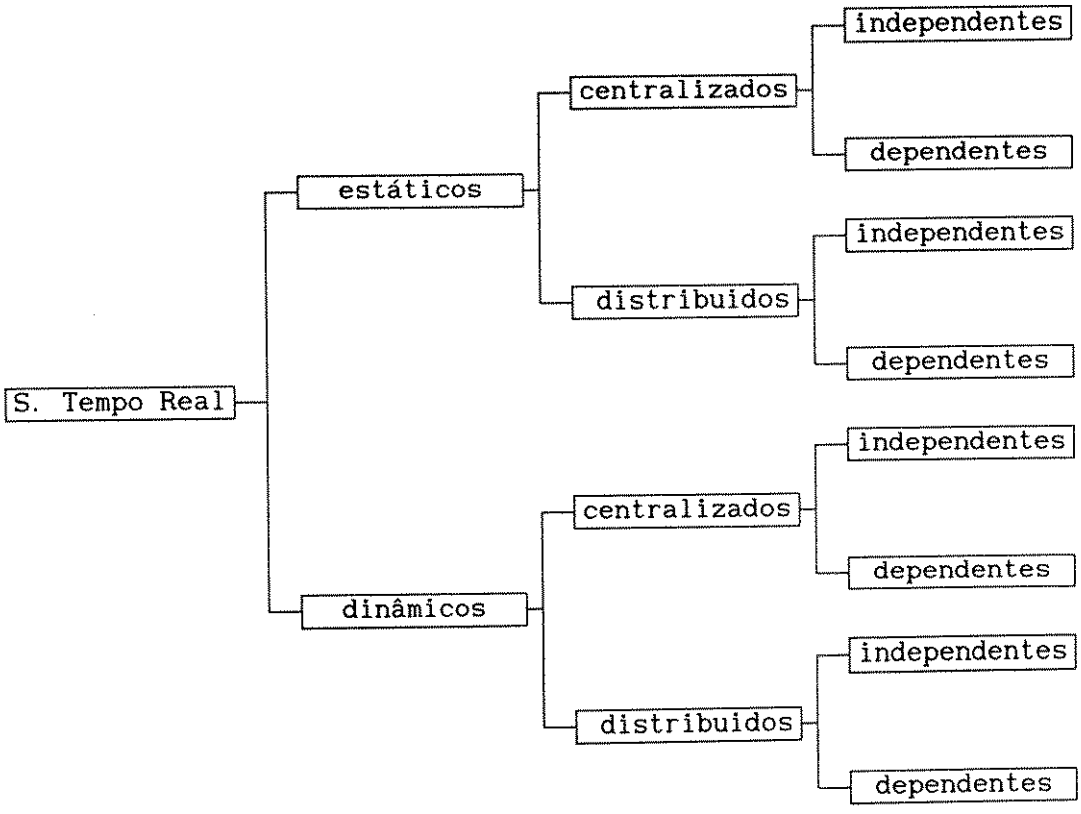

Figura 3.6. Classificação dos Sistemas de T. Real.

Neste trabalho estamos particularmente interessados nos sistemas centralizados, pois seus algoritmos podem também ser utilizados localmente nos sistemas distribuídos sendo portanto um "pré-requisito" para o seu desenvolvimento.

3.7. Alguns Resultados Associados ao STR

Neste item alguns dos algoritmos apresentados no item 3.6 para sistemas monoprocessadores serão explorados com o objetivo de delimitar claramente a faixa de atuação deste trabalho.

Nos próximos sub-itens vamos apresentar seis algoritmos utilizados em sistemas "Hard Real Time", sendo três direcionados para tarefas independentes e três para tarefas dependentes.

3.7.1. Algoritmo Taxa Monotônica

Taxa Monotônica é um algoritmo estático e centralizado (monoprocessador), que pode ser aplicado a tarefas periódicas, preemptíveis, independentes e que não necessitem de recursos específicos para executar [MAG90].

Sob este algoritmo, prioridades fixas são associadas às tarefas em função de seu período. Quanto mais requisitada for uma tarefa (i.e., menor o seu período), maior será a sua prioridade.

Este algoritmo executa a todo instante a tarefa no estado de pronto com maior prioridade.

TAREFAS PERIÓDICAS	C	P	D	PRIORIDADE
A	4	10	10	1
B	8	20	20	2

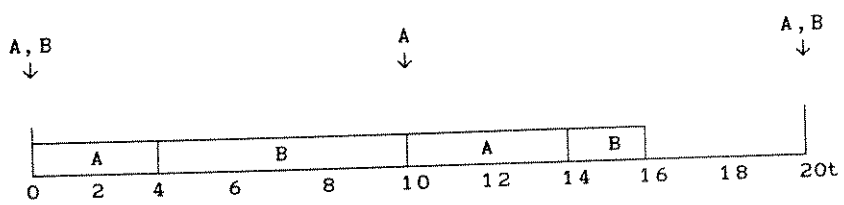

Figura 3.7. Exemplo do Algoritmo Taxa Monotônica.

Na Figura 3.7. atribui-se, em modo "off-line", as prioridades 1 e 2 às tarefas periódicas *A* e *B*, respectivamente, pelo fato do período (P) da tarefa *A* ser menor do que o período da tarefa *B*. O tempo de Computação (C) e o "Deadline" (D) das tarefas *A* e *B* também são representados na Figura 3.7..

Em t=0, as tarefas *A* e *B* iniciam seu primeiro período. Sendo a prioridade de *A* maior, ela é escalonada e ocupa o processador até t=4, quando completa a execução de sua primeira instância. Em t=4, sendo *B* a tarefa pronta de maior prioridade, esta é escalonada e executa 6 unidades até t=10, quando chega a segunda instância da tarefa *A*. A tarefa *B*, possuindo uma prioridade menor do que a tarefa *A*, sofre preempção dando lugar à tarefa *A* que executa até t=14, quando é completada. A tarefa *B* retoma a posse do processador e executa de t=14 até t=16 completando a sua primeira instância.

A sequência de execução das tarefas no intervalo t=20 a t=40 (t=40 a t=60, etc) comporta-se como no intervalo inicial t=0 a t=20. Deve ser observado que nenhum "deadline" é violado.

Liu [LIU73] prova que o algoritmo Taxa Monotônica é ótimo, no sentido de que nenhuma outra regra de associação pode escalonar um conjunto de tarefas periódicas que não possa ser escalonado pelo algoritmo Taxa Monotônica. Esse resultado é sintetizado pela Definição 1 e pelo Teorema 1 apresentados a seguir.

Definição 1. Fator de Utilização . O fator de utilização do processador representa a fração de tempo utilizada na execução do conjunto de tarefas. Seja C_i o tempo de computação da tarefa i e T_i o período da mesma tarefa, então o fator de utilização U, para um conjunto de n tarefas, é representado pela relação:

$$U=\sum_{i=1}^{n}(C_i/T_i)$$

Teorema 1. Um conjunto de n tarefas periódicas e independentes, escalonadas pelo algoritmo taxa monotônica, deve cumprir seu "deadline" se a relação abaixo for obedecida.

$$\frac{C_1}{T_1}+\ldots+\frac{C_n}{T_n}\leq n(2^{1/n}-1)=U(n)$$

O teorema 1 fornece um condição suficiente para determinar se um conjunto de tarefas pode ser escalonada pelo algoritmo Taxa Monotônica. Quando o número de tarefas tende a infinito a relação acima converge para 69% (*ln 2*).

O algoritmo apresentado tem aplicação imediata em diversas situações principalmente pelo fato de utilizar um esquema de prioridade fixa que é encontrado na maioria dos sistemas comerciais. Uma séria desvantagem desse algoritmo está relacionada com o tratamento a ser dado a tarefas não periódicas no sistema.

Para contornar esse problema, diversas extensões foram realizadas ao algoritmo Taxa Monotônica [SPR89]. O objetivo principal dessas extensões é garantir o "deadline" das tarefas periódicas e fornecer um tempo médio de resposta aceitável para as tarefas aperiódicas. Essas extensões serão apresentadas a seguir.

a) "Background Server" (BS).

Esta estratégia é uma das mais primitivas no tratamento das tarefas aperiódicas. Basicamente, ela consiste em aguardar os instantes nos quais o processador está desocupado para processar as tarefas aperiódicas. Se a carga imposta pelo conjunto de tarefas periódicas é alta, então a utilização liberada para as tarefas aperiódicas é baixa, resultando num alto tempo de resposta para tais tarefas. A Figura 3.8. apresenta um exemplo de utilização desta técnica. Note que o tempo de resposta das tarefas *C* e *D* é 12 e 6, respectivamente.

TAREFAS PERIÓDICAS	C	P	D	PRIORIDADE
A	4	10	10	1
B	8	20	20	2

TAREFAS APERIÓDICAS	C	P	D	PRIORIDADE
C	1	-	-	3
D	1	-	-	3

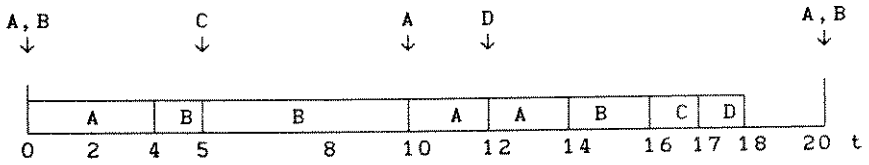

Figura 3.7. Exemplo do Algoritmo "Background Server".

b) "Polling Server" (PS).

Nesta estratégia é criada uma tarefa periódica servidora ("polling server") para atender as tarefas aperiódicas. A tarefa servidora periodicamente realiza uma análise para verificar se existem tarefas aperiódicas aguardando serviço; caso existam, essas tarefas são atendidas na medida da capacidade de processamento da tarefa servidora.

Dessa forma, o tempo médio de resposta das tarefas aperiódicas cai em média para a metade do período da tarefa servidora. A Figura 3.9 apresenta um exemplo de utilização desta estratégia indicando também a capacidade de processamento da tarefa "polling server".

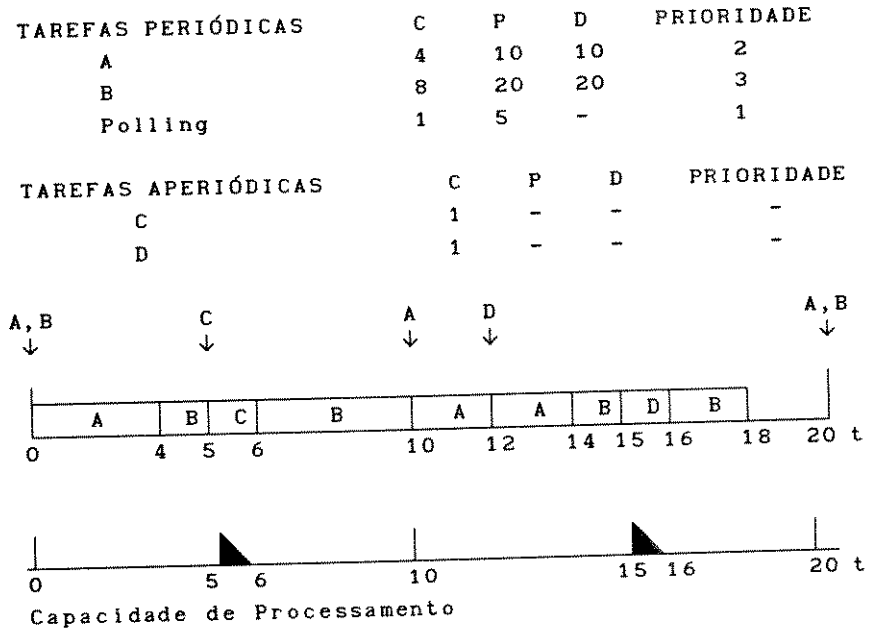

Figura 3.9. Exemplo do Algoritmo "Polling Server".

Na Figura 3.9. podemos observar que a capacidade de processamento da tarefa "polling server" é utilizada somente em t=5 e t=15 sendo desperdiçada nas outras ativações da servidora. O tempo de resposta para a tarefa *C* é imediato, entretanto, a tarefa *D* deve aguardar até o instante t=15 para ser atendida.

Este exemplo mostra que o tempo de resposta das tarefas aperiódicas melhorou em comparação com a estratégia "background server", mas nem sempre é possivel atender "imediatamente" a tais tarefas.

c) "Deferrable Server" (DS).

O algoritmo PS não preserva a capacidade de processamento da tarefa "polling", ou seja, ao ocorrer uma ativação da tarefa servidora e não existir nenhuma tarefa aperiódica necessitando de serviço, a capacidade de processamento é simplesmente descartada, como ocorre nos instantes t=0, t=10 e t=20 na Figura 3.9..

Para contornar essa deficiência, a estratégia "deferrable server" mantém armazenada a capacidade de processamento não utilizada pela tarefa servidora. Essa técnica melhora o tempo médio de resposta das tarefas aperiódicas por seu potencial de atender imediatamente requisições de tarefas aperiódicas sempre que existir capacidade de processamento armazenada.

A Figura 3.9. apresenta um exemplo de utilização do algoritmo DS bem como indica a capacidade de processamento mantida pela tarefa servidora.

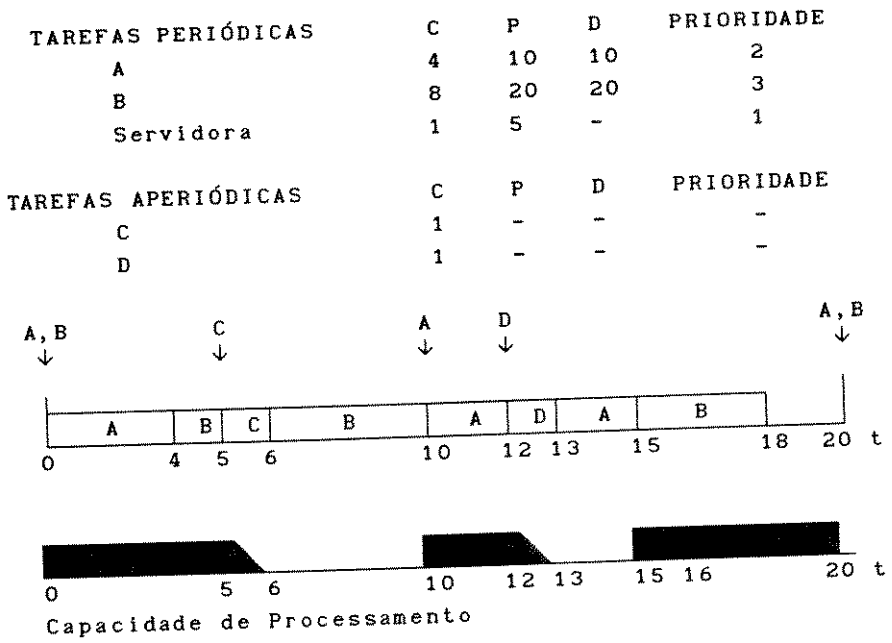

Figura 3.10. Exemplo do Algoritmo "Deferrable Server"

Observe que, no exemplo descrito pela Figura 3.10., apesar da ativação da tarefa *D* ocorrer no instante t=12, que não coincide com o período da tarefa servidora, ela é ativada imediatamente melhorando seu tempo de resposta em relação à estratégia PS.

d) "Priority Exchange" (PE).

O algoritmo "Deferrable Server" mantém a capacidade de processamento do servidor na mesma prioridade, mesmo que não existam tarefas aperiódicas para executar. Essa estratégia, segundo Sprunt, Sha e Lehoczky [SPR89], diminui o fator de utilização do processador U (definição 1).

O algoritmo "Priority Exchange" minimiza esse problema cedendo sua capacidade de processamento não utilizada para tarefas com prioridades inferiores. No início do período da tarefa servidora sua capacidade de processamento é elevada a sua capacidade máxima. Caso existam tarefas aperiódicas pendentes elas são processadas nesse instante. Caso não existam tarefas aperiódicas pendentes o tempo de execução reservado para tais tarefas é liberado para execução da tarefa periódica com maior prioridade presente no sistema. Todo o tempo cedido para a tarefa periódica é contabilizado a favor das tarefas aperiódicas, só que, na prioridade da tarefa periódica que o consumiu.

A Figura 3.11. apresenta um exemplo de utilização deste algoritmo.

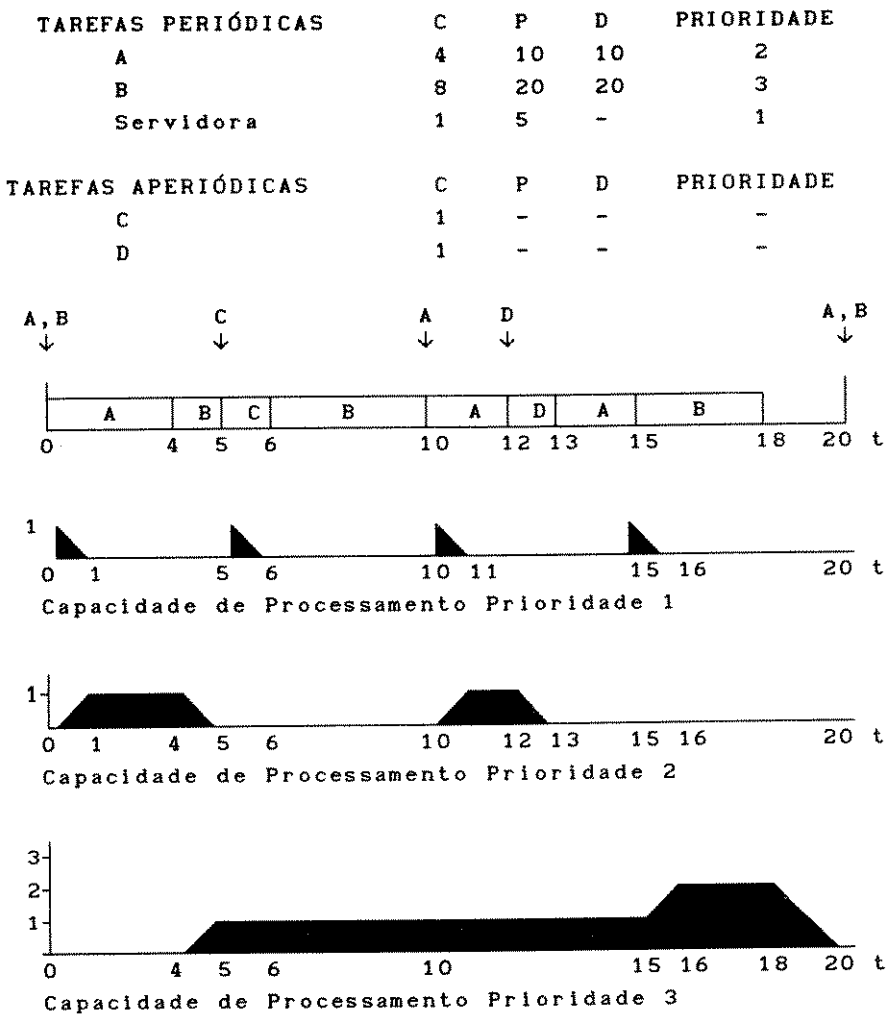

Figura 3.11. Exemplo do Algoritmo "Priority Exchange Server"

No instante t=0 o servidor de tarefas é ativado com sua capacidade máxima; como não existem tarefas aperiódicas para executar ocorre uma troca de prioridade entre os níveis 1 e 2. O servidor de tarefas acumula tempo de processamento na prioridade 2, e a tarefa periódica *A* é executada na prioridade 1. No instante t=4, a tarefa *A* encerra seu processamento e cede o processador para a tarefa *B*. Como ainda não existem tarefas aperiódicas pendentes ocorre outra troca de prioridades entre os níveis 2 e 3.

No instante t=5 a capacidade de execução do servidor de tarefas na prioridade 1 é novamente elevada a seu valor máximo sendo utilizada para atender a tarefa aperiódica *C*.

No instante t=10 outra ativação do servidor eleva novamente a capacidade de processamento da prioridade 1 a seu máximo e, como não existem tarefas

aperiódicas para executar ocorre nova troca de prioridade com o nível 2. No instante t=12 ocorre a ativação da tarefa aperiódica *D* que é então executada na prioridade 2.

No instante t=15 a capacidade de processamento do servidor é reabastecida. Como a tarefa periódica mais prioritária em condições de executar é a tarefa *B*, o tempo reposto é repassado para a prioridade 3.

No instante t=18 toda a capacidade armazenada na prioridade 3 é esvaziada pois não existem tarefas prontas para executar.

e) "Sporadic Server" (SS).

Nesta estratégia ao invés de atualizar a capacidade de processamento do servidor periodicamente, em pontos fixos no tempo, a reposição somente é determinada quando algum ou todo o tempo de execução for consumido.

Para apresentar o algoritmo SS utilizaremos a mesma notação adotada por Sprunt, Sha e Lehoczky [SPR89]. Os seguintes termos são necessários nessa apresentação:

P_s - O nível de prioridade no qual o sistema está executando.

P_i - Um dos níveis de prioridade do sistema. Os níveis de prioridade são numerados de acordo com sua prioridade, ou seja, P_1 possui a mais alta prioridade, P_2 vem a seguir, e assim sucessivamente até o último.

ativado - Descreve um período de tempo em relação a uma prioridade particular. Um nível de prioridade P_i está ativo se a prioridade atual do sistema P_s é igual ou maior do que P_i.

desativado - Tem o significado oposto do anterior. Um nível de prioridade P_i está desativado se a prioridade corrente do sistema P_s é menor do que a prioridade P_i.

RT_i - O instante de reposição para o nível de prioridade P_i. Este é o momento no qual o tempo de execução consumido pelo "Sporadic Server" de nível P_i é reposto.

A "chave" para compreender o algoritmo SS consiste em analisar como ocorre a reposição de tempos consumidos pelas rotinas aperiódicas.

Essa reposição é dividida em dois momentos distintos. No primeiro, é levantado o instante da reposição que é sempre igual ao instante atual somado ao período do "sporadic server".

No segundo, o instante de reposição levantado anteriormente é validado autorizando ou não sua ocorrência. Estes dois momentos são descritos formalmente nos próximos parágrafos.

Se o servidor possui tempo de execução disponível, o instante de reposição RT_i é atualizado quando o nível de prioridade P_i torna-se ativo. O valor de RT_i assume um valor igual à soma do tempo corrente mais o período de P_i.

Se a capacidade do servidor se esgotou o instante de reposição RT_i é atualizado quando a capacidade de processamento do servidor assume um valor diferente de zero e P_i torna-se ativo.

O instante de reposição é validado se o nível de prioridade P_i torna-se inativo e parte do tempo de execução foi consumido, ou se todo o tempo de execução do servidor foi consumido. O valor a ser reposto é igual à quantidade de tempo de execução consumida pelo servidor.

Para uma visão mais concreta deste algoritmo vamos apresentar dois exemplos de utilização obtidos em [SPR89], o primeiro alocando um nível de prioridade ao servidor igual ao nível da tarefa periódica mais prioritária no sistema e, o segundo, alocando um nível de prioridade intermediário. Nesses exemplos as tarefas aperiódicas possuem 1 unidade de tempo de execução.

A Figura 3.12. descreve a execução das tarefas para o primeiro exemplo. O servidor e a tarefa *A* executam no mesmo nível de prioridade P_1, a tarefa *B* executa no nível de prioridade P_2.

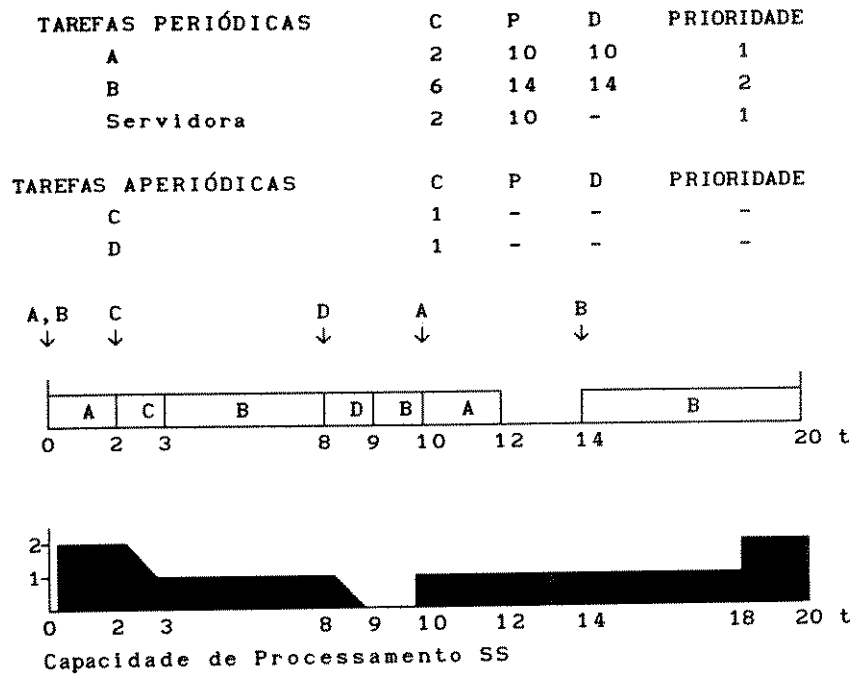

Figura 3.12. Exemplo do Algoritmo SS

No instante t=0 a tarefa *A* inicia a execução, P_1 torna-se ativo e RT_1 recebe o valor 10.

No instante t=2 ocorre a primeira requisição da tarefa aperiódica *C* que é atendida. Em t=3 a tarefa aperiódica completa sua execução liberando o processador para a tarefa *B*; com isso, P_1 torna-se inativa validando a reposição RT_1 no instante t=10 com o valor 1.

No instante t=8 ocorre a segunda requisição de tarefa aperiódica, que é atendida levando P_1 ao estado ativo e RT_1 recebe um valor igual à 18. No instante t=9 encerra-se a execução da tarefa aperiódica *D*, P_1 torna-se inativo validando a reposição RT_1 no instante t=18 com o valor de 1.

Em t=10 a tarefa *A* é ativada, P_1 torna-se ativo e RT_1 assume o valor 20. No instante t=12 *A* encerra sua execução e P_1 torna-se inativo. Entretanto, como não houve consumo do tempo de execução do servidor o instante de reposição RT_1 não é validado.

Este exemplo ilustra uma importante característica do algoritmo SS, que é de somente repor a capacidade de processamento do servidor pelo valor efetivamente consumido, ao contrário das estratégias DS e PE cuja reposição é sempre realizada com o valor máximo.

caso uma prioridade média para o servidor; dessa forma, o servidor executa entre os níveis P_1 e P_3 das tarefas *A* e *B*, respectivamente.

TAREFAS PERIÓDICAS	C	P	D	PRIORIDADE
A	1.0	5	5	1
B	6.0	14	14	3
Servidora	2.5	10	-	2

TAREFAS APERIÓDICAS	C	P	D	PRIORIDADE
C	1	-	-	-
D	1	-	-	-

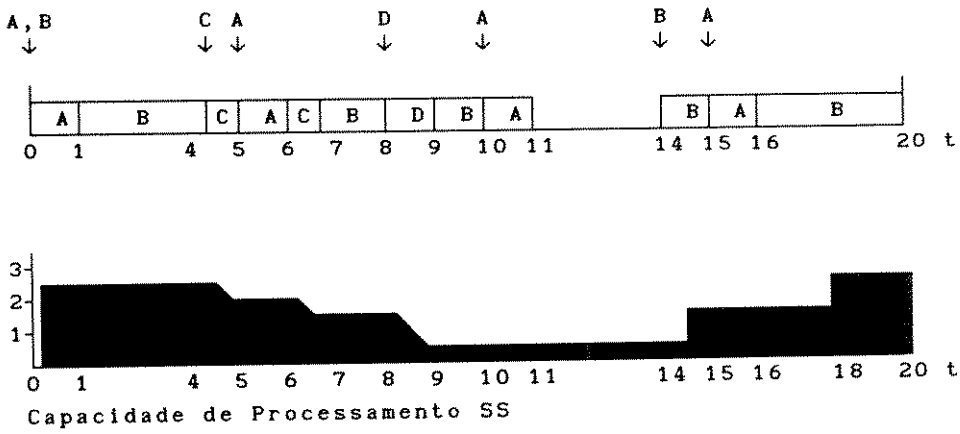

Figura 3.13. Exemplo do Algoritmo SS com prioridade média

No instante t=0 a tarefa *A* inicia sua execução ativando o nível de prioridade P_2 e programando a próxima reposição RT_2 para t=10. Em t=1 a tarefa *A* encerra sua execução cedendo o processador para a tarefa *B* e desativando P_2. Note que apesar de P_2 ser desativado RT_2 não é validado pois nenhum tempo de execução do servidor foi consumido.

No instante t=4,5 a primeira requisição aperiódica ocorre, recebendo o processador e ativando P_2, programando a reposição RT_2 para t=14,5. Em t=5 a tarefa *A* torna-se ativa e realiza preempção sobre o servidor. Nesse momento tanto P_1 quanto P_2 estão ativados.

No instante t=6 a tarefa *A* encerra sua execução e P_1 torna-se inativo liberando novamente o processador para o servidor. Em t=6,5 a tarefa aperiódica *C* encerra sua execução, a tarefa *B* continua executando e P_2 torna-se inativo. Nesse momento a reposição RT_2 é validada com o tempo a ser reposto igual ao consumido pela tarefa aperiódica *C*, ou seja, 1.

No instante t=8 a segunda requisição aperiódica ocorre e consome uma unidade da capacidade de processamento do servidor sendo então validada sua reposição em t=18.

Em t=9 a tarefa B continua sua execução até encerrar seu processamento em t=10. Nesse momento ocorre a segunda ativação da tarefa A que assume o processador até t=11 quando encerra sua execução.

O ciclo das tarefas periódicas continua normalmente a partir desse ponto sem ser alterado por novas ativações de tarefas aperiódicas.

Este exemplo ilustra outra importante propriedade do algoritmo SS; mesmo que o servidor seja interrompido em sua execução por uma tarefa de prioridade mais alta (como ocorre em t=5 na Figura 3.12.) somente um RT_i é necessário, ou seja, a preempção do servidor não faz com que seu nível de prioridade torne-se inativo, não permitindo o cálculo de um novo valor para RT.

Sob o ponto de vista de implementação o algoritmo SS possui menor complexidade do que o algoritmo PE, pois ele não comuta a capacidade de processamento do servidor com os níveis mais baixos de prioridade como necessita o algoritmo PE.

O algoritmo SS também possui vantagens em tempo de execução, pois ao contrário das estratégias DS e PE que repõem o tempo de execução do servidor periodicamente, a estratégia SS somente realiza a reposição se houver consumo.

As estratégias DS, PE e SS utilizam o algoritmo Taxa Monotônica para atribuir prioridades às tarefas periódicas e ao servidor; entretanto, o fator de utilização do processador U alcançado é distinto para cada uma.

Para definir o fator de utilização do processador para as estratégias DS e PE é necessário definir previamente a quantidade de utilização do processador reservada para tratar tarefas aperiódicas. Esse valor é denominado "server size" (U_s) e representa a razão entre o tempo de execução do servidor e seu período, ou seja:

$$U_s = \frac{tempo\ de\ execução}{período}$$

O "server size" U_s e o tipo de algoritmo (PE ou DS) determinam o limite de escalonabilidade para tarefas periódicas U_p, que representa a maior utilização do processador para a qual a estratégia Taxa Monotônica garante o escalonamento das tarefas periódicas.

O limite U_p pode ser representado em termos do "server size" U_s, quando o número de tarefas periódicas tende ao infinito, pelas seguintes equações [SPR89]:

$$PE:\ U_p = ln\ \frac{2}{U_s + 1} \qquad\qquad DS:\ U_p = ln\ \frac{U_s + 2}{2U_s + 1}$$

Essas equações mostram que para o mesmo "server size", o limite de escalonabilidade U_p é menor para o algoritmo DS do que para o algoritmo PE.

Para o algoritmo "sporadic server" o mesmo fator de utilização obtido no *teorema* 1 aplicando a estratégia Taxa Monotônica sobre tarefas periódicas e independentes é válido, conforme demonstrado por Sprunt, Sha e Lehoczky [SPR89]. Dessa forma, o servidor do algoritmo "sporadic server" possui o mesmo comportamento que uma tarefa periódica comum.

3.7.2. Algoritmo "Earliest Deadline First"

Vamos agora analisar um dos algoritmos de escalonamento dinâmico mais conhecidos e simples de implementar: o algoritmo "Earliest Deadline First".

A base deste algoritmo consiste em atribuir prioridades para as tarefas de acordo com o "deadline" de sua ativação corrente. Dessa forma, a tarefa com a prioridade mais crítica possui o "deadline" da ativação corrente mais próximo de "estourar". A qualquer instante a tarefa com o "deadline" mais crítico está ocupando o processador.

Este algoritmo também é apresentado no trabalho de Liu [LIU73] que provou o teorema 2 apresentado a seguir.

Teorema 2. Para um conjunto de *n* tarefas periódicas , o algoritmo "Earliest Deadline First" atende todos os "deadlines" se e somente se:

$$U = \sum_{i=1}^{n} (C_i/T_i) \leq 1$$

A Figura 3.14. apresenta um exemplo deste algoritmo utilizando duas tarefas periódicas *A* e *B*.

TAREFAS PERIÓDICAS	C	P	D
A	1	4	4
B	3	5	5

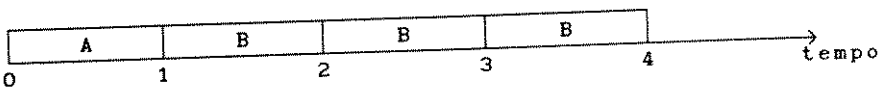

Figura 3.14. Exemplo do Algoritmo "Earliest Deadline First"

No instante *t*=0 ocorre a ativação das tarefas *A* e *B*; como a tarefa *A* possui o "deadline" mais crítico, ela é escolhida para entrar em execução.

Em t=1 a tarefa *A* encerra sua execução, liberando o processador para a tarefa *B* que encerra sua execução em *t*=4.

3.7.3. Algoritmo "Least Laxity First"

Este algoritmo também é ótimo para tarefas periódicas e independentes num sistema dinâmico com preempção permitida, como o algoritmo "Earliest Deadline First" apresentando o mesmo fator de utilização do processador *U* como demonstrado por Mok [MOK84].

A estratégia neste algoritmo consiste em ativar a tarefa com a menor "folga" de processamento até o estouro de seu "deadline" ou, em outras palavras, ativar a tarefa com o menor { d(t) - t - c(t) }, onde d(t) representa o "deadline" corrente e c(t) representa o tempo de computação restante no instante *t*.

A Figura 3.15. apresenta um exemplo de utilização do algoritmo "Least Laxity First" identificando o valor de (d(t) - t - c(t)) a cada instante do processamento.

TAREFAS PERIÓDICAS	C	P	D
A	1	4	4
B	3	5	5

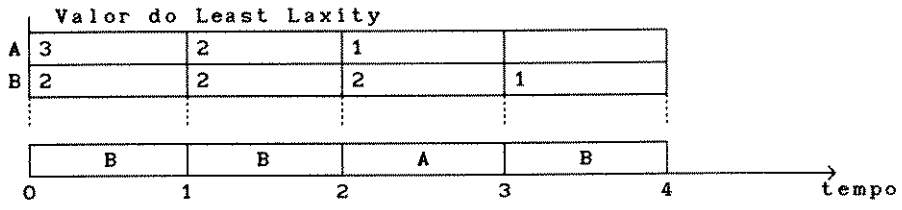

Figura 3.15. Exemplo do Algoritmo "Least Laxity First"

Em t=0 ocorre a ativação das tarefas *A* e *B*, como o valor "Least Laxity" (LL) da tarefa *B* é menor que o valor LL da tarefa *A*, ela é escolhida para entrar em execução. A tarefa *B* continua em execução até t=2 quando então, é substituida pela tarefa *A* cujo valor LL tornou-se menor que o valor da tarefa *B*. No instante t=3 a tarefa *A* encerra seu processamento cedendo o processador para a tarefa *B* que executa até seu final em t=4.

3.7.4. Protocolo de Limite de Prioridade.

Os três algoritmos apresentados nos itens anteriores são direcionados para tarefas independentes, pois assumem que qualquer processo pronto para executar pode ser livremente selecionado e efetuar preempção sobre outro processo [MOK84]. Entretanto, na maioria das aplicações as tarefas são dependentes e realizam sincronização e comunicação entre si.

Os mecanismos tradicionais para sincronização e comunicação de tarefas como semáforos, "mailbox" e "rendezvous" da linguagem ADA não são apropriados para sistemas HTR. Mok [MOK84] provou que o problema de decidir se é possível escalonar um conjunto de processos periódicos é NP-hard quando utilizam semáforos para garantir exclusão mútua.

Dessa forma, soluções alternativas foram pesquisadas das quais podemos destacar duas linhas. A primeira consiste em inserir restrições na utilização das técnicas de sincronização; nesta classe se encontram o monitor e o "rendezvous determinístico" desenvolvidos por Mok, além das técnicas de herança de prioridade [SHA90b] .

A segunda linha utiliza algoritmos heurísticos para gerar escalonamentos válidos obtendo altas probabilidades de sucesso nessa geração. Um exemplo de utilização de algoritmos heurísticos pode ser encontrado no trabalho de Zhao, Ramamritham e Stankovic [ZHA087b].

Neste item e nos seguintes vamos apresentar alguns dos algoritmos propostos para tratar sistemas uniprocessadores com tarefas dependentes.

Protocolo de Limite de Prioridade [SHA90b] é uma técnica aplicada ao algoritmo Taxa Monotônica para permitir interação entre tarefas. Nos sistemas com multiprogramação tradicionais os recursos clássicos para permitir sincronização e comunicação entre tarefas, como, por exemplo, semáforos, mailboxes e monitores, criam situações que inviabilizam as estratégias adotadas para sistemas "Hard Real Time".

A Figura 3.16. apresenta um conjunto de tarefas { T_1, T_2, T_3}, sendo que T_1 e T_3 mantêm uma região crítica controlada pelo semáforo S_1.

TAREFAS	C	P	D	TEMPO DE PRONTO
T1	10	40	40	4
T2	10	50	50	8
T3	10	60	60	0

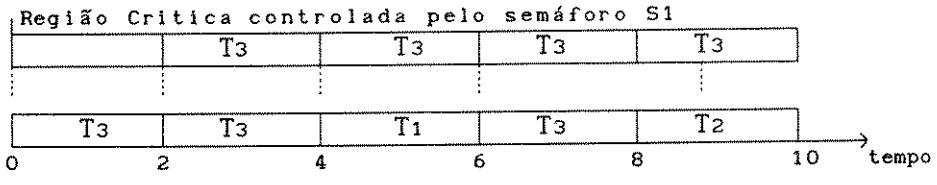

Figura 3.16. Sincronização através de Semáforos.

No instante t=0 a tarefa T_3 é ativada e entra em execução, em t=2 a tarefa T_3 entra na região crítica controlada pelo semáforo S_1, em t=4 a tarefa T_1 é criada e como possui maior prioridade, entra em execução; em t=6 a tarefa T_1

tenta acessar a região crítica, sendo suspensa pelo fato de T_3 já estar acessando.

No instante t=8 a tarefa T_2 é ativada realizando preempção sobre T_3, recebendo o processador. Dessa forma, a tarefa T_2, que não acessa a região crítica controlada por S_1 retarda a tarefa T_1 indiretamente.

Infelizmente, a aplicação direta de mecanismos de sincronização pode conduzir a situações como a apresentada no exemplo acima onde tarefas com baixa prioridade inibem a execução de uma tarefa de alta prioridade, por um período de tempo indeterminado, caracterizando uma inversão de prioridade.

Neste item apresentaremos uma técnica para controlar o efeito de inversões de prioridade denominada protocolo de limite de prioridade. Uma descrição formal desse protocolo, bem como a prova dos resultados apresentados pode ser obtida no trabalho de Sha, Rajkumar e Lehoczky [SHA90b].

O protocolo de limite de prioridade é um mecanismo de sincronização com duas propriedades fundamentais:

1. livre de "deadlock",
2. inversão de prioridade controlada, ou seja, no máximo uma tarefa de baixa prioridade pode bloquear uma tarefa de alta prioridade.

Para implementar essa estratégia é necessário inicialmente determinar o limite de prioridade de cada semáforo, que é igual à maior prioridade de todas as tarefas que o utilizam.

A idéia básica do protocolo é garantir que quando uma tarefa T_i realiza preempção sobre a região crítica de outra tarefa, e executa sua própria região crítica z_i, a prioridade na qual a região crítica z_i será executada é garantidamente maior que o limite de prioridade de todas as regiões críticas que sofreram preempção. Se esta condição não pode ser satisfeita a tarefa T_i é suspensa, e a tarefa que inibiu a execução de T_i herda sua prioridade.

Assim, uma tarefa T_i somente pode iniciar a execução de uma região crítica se sua prioridade é maior do que todos os limites de prioridade dos semáforos bloqueados por outras tarefas.

O próximo exemplo ilustra melhor a estratégia apresentada acima. Suponha três tarefas T_0, T_1 e T_2 em ordem decrescente de prioridade. Suponha também que temos duas regiões críticas controladas pelos semáforos S_1 e S_2 com a sequência de passos de processamento descritos abaixo:

T_0 = { ...,**P**(S_0),...,**V**(S_0),... }
T_1 = { ...,**P**(S_1),...,**P**(S_2),...,**V**(S_2),...,**V**(S_1),... }
T_2 = { ...,**P**(S_2),...,**P**(S_1),...,**V**(S_1),...,**V**(S_2),... }

A Figura 3.17. apresenta o comportamento das tarefas fornecidas considerando-se semáforos clássicos. A sequência de eventos é descrita a seguir.

- No instante t_0 a tarefa T_2 é ativada e inicia sua execução realizando a operação $P(S_2)$.
- No instante t_1 a tarefa T_1 é ativada e realiza preempção sobre a tarefa T_2.
- Em t_2 a tarefa T_1 acessa a região crítica controlada pelo semáforo S_1.
- No instante t_3 a tarefa T_0 é ativada e assume o processador por possuir a maior prioridade. Posteriormente T_0 acessa a região crítica controlada pelo semáforo S_0.
- Em t_4 a tarefa T_0 encerra seu processamento cedendo o processador para a tarefa T_1 que continua acessando a região crítica controlada por S_1.
- No instante t_5 a tarefa T_1 tenta acessar a região crítica controlada por S_2 que já está ocupada pela tarefa T_2. Dessa forma, T_1 é bloqueada cedendo o processador para T_2.
- No instante t_6 a tarefa T_2 tenta acessar a região crítica controlada pelo semáforo S_1 que está ocupado pela tarefa T_1. Dessa forma, T_2 é bloqueada à espera da liberação de S_1. Nesse instante fica caracterizado o "deadlock" onde nem T_1 e nem T_2 podem executar.

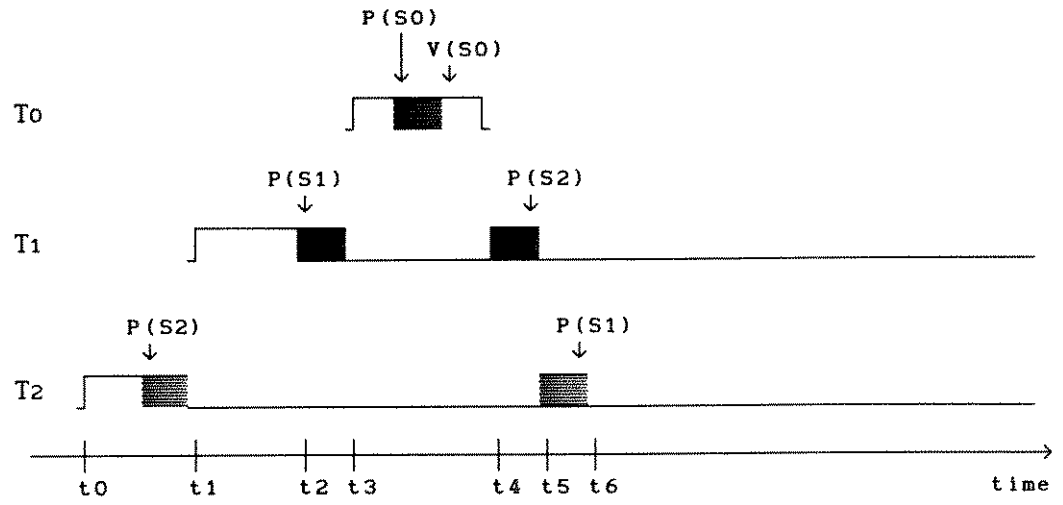

Figura 3.17. Exemplo de "deadlock" com semáforos clássicos.

A figura 3.18. representa o comportamento das tarefas considerando o protocolo Limite de Prioridade. De acordo com nossa definição, o limite de prioridade dos semáforos S_1 e S_2 é a prioridade da tarefa T_1 e o limite de prioridade do semáforo S_0 é a prioridade da tarefa T_0.

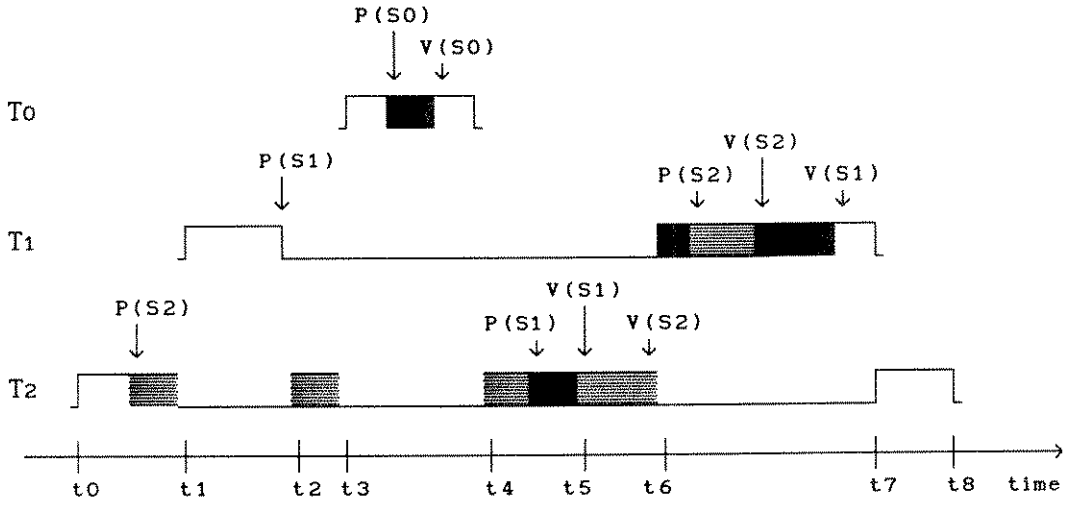

Figura 3.18. Exemplo de Herança de Prioridade.

A sequência de eventos que ocorrem na Figura 3.18. é descrita a seguir:

- No instante t_0, T_2 é ativado e inicia sua execução realizando a operação $P(S_2)$.
- No instante t_1 a tarefa T_1 é ativada e realiza preempção sobre a tarefa T_2.

- No instante t_2 a tarefa T_1 tenta entrar na região crítica controlada pelo semáforo S_1, entretanto a prioridade da tarefa T_1 não é maior do que a do semáforo ocupado S_2, com isso a tarefa T_1 é suspensa sem acessar a região crítica controlada pelo semáforo S_1. A tarefa T_2, por sua vez, herda a prioridade da tarefa T_1 e continua sua execução.
- No instante t_3, T_2 ainda se encontra na sua região crítica quando a tarefa T_0 é ativada assumindo o processador. Posteriormente T_0 acessa a região crítica controlada pelo semáforo S_0.
- Em t_4, T_0 encerra seu processamento cedendo o processador para a tarefa T_2, desde que T_1 foi bloqueada por T_2 e não pode executar, T_2 continua sua execução e acessa a região crítica controlada por S_1.
- Em t_5, T_2 sai da região crítica controlada por S_1.
- No instante t_6, T_2 libera S_2 e retorna a sua prioridade original. Com isso, T_1 é ativado por possuir uma prioridade maior que T_2. T_1 acessa então as regiões críticas controladas pelos semáforos S_1 e S_2.
- Em t_7, T_1 encerra sua execução liberando o processador para a tarefa T_2.
- No instante t_8, T_2 encerra sua execução.

O Teorema 3. apresenta um resultado importante relacionado com a utilização do protocolo de limite de prioridade. Sua demonstração pode ser encontrada no trabalho de Sha [SHA90b].

Teorema 3. Um conjunto de n tarefas periódicas utilizando o protocolo de limite de prioridade pode ser escalonado pelo algoritmo Taxa Monotônica se a seguinte condição é satisfeita:

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} + \max\left(\frac{B_1}{T_1}, \ldots, \frac{B_{n-1}}{T_{n-1}}\right) \leq n(2^{1/n} - 1) = U(n)$$

Onde B_i representa o tempo do pior caso de bloqueio ocorrido sobre a tarefa T_i.

O protocolo Limite de Prioridade associado ao escalonamento Taxa Monotônica representa uma ferramenta simples e poderosa no desenvolvimento de sistemas "Hard Real Time". Como forma de aprofundar a análise deste protocolo, ele foi implementado em uma das versões do núcleo apresentado no próximo capítulo.

3.7.5. "Rendezvous" Determinístico.

A utilização de semáforos para sincronizar tarefas normalmente gera códigos não estruturados facilitando a geração de erros.

Mok [MOK84] desenvolveu um modelo baseado em troca de mensagens, derivado da proposta inicial desenvolvida para comunicação entre tarefas através da primitiva "rendezvous" presente na linguagem ADA.

Antes de apresentar o modelo proposto é necessário definir alguns dos termos utilizados nos resultados apresentados. Para propósitos de escalonamento, uma tarefa T_i consiste de uma cadeia de *blocos de escalonamento*, $\{ T_{i,j}, j=1,n_i \}$ onde $T_{i,j}$ representa um trecho de código executado após $T_{i,j-1}$.

Cada $T_{i,j}$ possui um tempo de computação $c_{i,j}$ conhecido à priori, sendo que, a soma dos $c_{i,j}$ representa o tempo total de computação c_i da tarefa T_i. Coordenação entre tarefas é obtida através de primitivas de comunicação que somente podem ser utilizadas entre os blocos de escalonamento.

Quando uma tarefa está pronta para executar, por exemplo no tempo t, ela deve encerrar seu processamento dentro de um tempo limite denominado "deadline" (d_i) relativo ao tempo t, ou seja, o último bloco de escalonamento de T_i deve completar sua execução antes de $t+d_i$.

Mok também define dois tipos de tarefas: periódicas e esporádicas. Se T_i é periódica, ela é ativada a cada p_i intervalo de tempo iniciando no tempo 0. Se T_i é esporádica, ela pode ser ativada a qualquer instante, com a restrição de que ativações consecutivas possuem um intervalo de no mínimo p_i unidades de tempo.

Formalmente, uma instância do modelo desenvolvido por Mok $M = M_p \cup M_s$ é um conjunto finito de tarefas representadas através da união de subconjuntos disjuntos: M_p (as tarefas periódicas) e M_s (as tarefas esporádicas).

Uma tarefa T_i possui três parâmetros: c_i (tempo de computação), d_i ("deadline") e p_i (período) onde $c_i \leq d_i \leq p_i$. Se uma tarefa $T_i \in M_p$ então ela é ativada a todo instante kp_i, $k \in \mathbb{N}$. Se a tarefa $T_i \in M_s$ então ela pode ser ativada a qualquer instante t , com a condição de que duas ativações sucessivas devem ter um intervalo mínimo de p_i unidades de tempo.

Segundo Mok o problema de escalonamento de sistemas HTR envolve dois escalonadores: um *off-line* e um *run-time*. O escalonador "off-line" analisa as tarefas que compõem o sistema e cria o escalonador "run-time" associado a uma base de dados capaz de tomar decisões em tempo de execução.

Com esses elementos definidos podemos agora retomar a técnica de "rendezvous" determinístico que é o objetivo deste item. Quando uma tarefa T_i executa um "rendezvous" ela deve aguardar até que a tarefa T_j ,que é alvo do "rendezvous", também execute uma primitiva de "rendezvous" direcionada para a tarefa T_i.

O próximo exemplo mostra que o algoritmo "Earliest Deadline First" alterado para ativar a tarefa pronta para executar com o "deadline" mais crítico e não bloqueada por um "rendezvous", não é factível.

Suponha três tarefas periódicas: T_1, T_2 e T_3. T_1 consiste de dois blocos de escalonamento com $c_{11}=c_{12}=1$, $d_1=3$ e $p_1=5$. T_2 possui dois blocos de escalonamento com $c_{21}=1$, $c_{22}=3$ e $d_2=p_2=10$. Finalmente T_3 possui somente um bloco de escalonamento com $c_3=1$, $d_3=9$ e $p_3=10$. T_1 mantém um "rendezvous" com T_2 após o seu primeiro bloco de escalonamento, e T_2 mantém "rendezvous" com T_1 após seu primeiro e segundo blocos de escalonamento (Figura 3.19.).

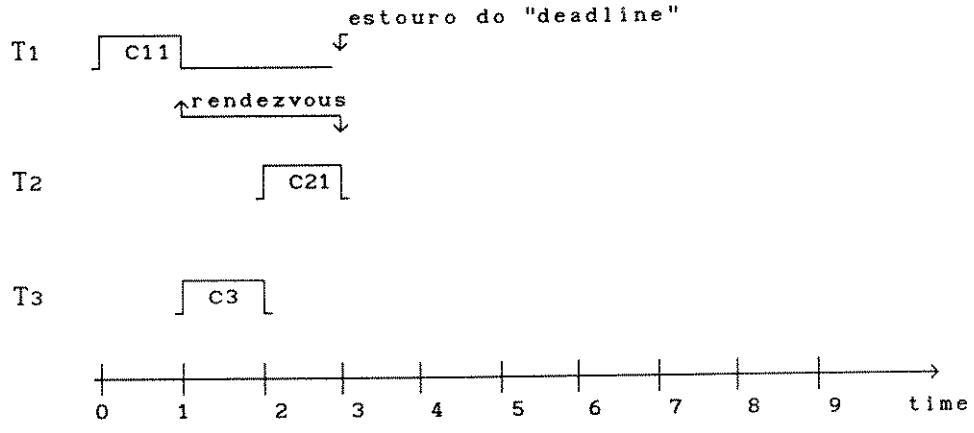

Figura 3.19. Estouro de "deadline" da tarefa T_1 com "rendezvous"

O algoritmo "Earliest Deadline First" falha porque ele não faz uso da informação de que T_2 deve manter dois "rendezvous" com T_1, ou seja, o segundo "rendezvous" de T_2 deve acabar antes do final do segundo "deadline" de T_1. Dessa forma, o "deadline" real de T_2 é $d_2=7$ ao invés de 10 como

especificado originalmente; com esse novo "deadline" T_2 torna-se mais prioritária que T_3 e será executada primeiro.

A estratégia desenvolvida por Mok, basicamente, revisa os "deadlines" de cada bloco de escalonamento construindo uma base de dados para o escalonador "run-time" tal que o algoritmo EDF possa ser utilizado. O teorema 4 apresenta esse resultado.

Teorema 4. Se existe um escalonamento factível para um conjunto de tarefas que utilizam a primitiva "rendezvous", então esse conjunto pode ser escalonado pelo algoritmo "Earliest Deadline First", alterado para escalonar o processo pronto para executar, que não esteja bloqueado por um "rendezvous", e possua o "deadline" revisado mais próximo.

3.7.6. O modelo "Kernelized Monitor"

Para modelar regiões críticas Mok [MOK84] propõe a utilização de monitores [HOA74] com um tratamento especial realizado pelo núcleo. É necessário ressaltar que o problema de escalonamento com monitores é NP-hard; entretanto, o problema pode ser equacionado se o tempo de duração de cada ativação do monitor for rigidamente controlado.

Neste modelo o sistema operacional garante a exclusão mútua entre tarefas através da alocação do tempo do processador somente em partes não interruptíveis de tamanho q ("quantum"), que é maior que o tempo consumido pelo monitor mais longo.

O modelo também considera que o tempo de computação c_i de todos os blocos de escalonamento é um múltiplo exato de q. Esta restrição é razoável se todas as regiões críticas são pequenas; por exemplo, acessam um conjunto pequeno de variáveis que devem ser mantidas mutuamente consistentes.

O próximo exemplo mostra que se aplicarmos diretamente o algoritmo EDF para escalonar um sistema com monitores, podemos não obter o escalonamento factível. Esse exemplo ilustra ainda o conceito de *regiões proibidas* que será utilizado na descrição desta técnica.

Suponha duas tarefas periódicas T_1 e T_2. T_1 consiste de uma única região crítica de tamanho $c_1=2$, "deadline" $d_1=2$ e período $p_1=5$. T_2 possui dois blocos de escalonamento com as seguintes características: $c_{21}=2$, $c_{22}=2$, $p_2=d_2=10$, sendo que o segundo bloco de escalonamento é a mesma região crítica utilizada por T_1. Dessa forma o intervalo de preempção é igual a 2 (Figura 3.20.).

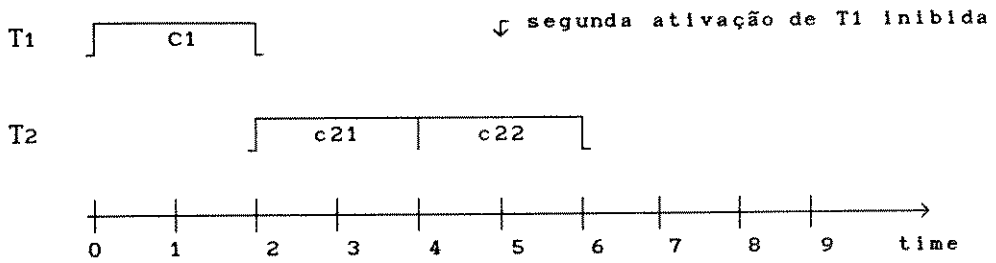

Figura 3.20. Estouro do "deadline" da tarefa T_1 com monitor

O segundo "deadline" de T_1 é perdido em t=7 pois o bloco de escalonamento c_{22} inibe sua ativação em t=5. Uma forma de contornar esse problema é deixar o processador livre no intervalo [4,5] e executar c_{22} em [7,9]. O intervalo aberto (4,5) é um exemplo de região proibida, na qual o escalonador não deve alocar o "quantum" de execução a nenhuma tarefa, para garantir que um "deadline" futuro seja garantido.

A base de dados utilizada pelo escalonador "run-time" mantém todas as regiões proibidas no intervalo $[0,L]$, onde L é o maior período entre as tarefas que fazem parte do sistema. O escalonador "run-time" atualiza a base de dados a todo intervalo L, localizando todas as regiões proibidas futuras.

A idéia básica do escalonador baseado no "kernelized monitor" é: Seja t o instante no qual o processador está desocupado e não pertence a uma região proibida; o escalonador aloca o próximo "quantum" do processador para a tarefa pronta para executar que possui o "deadline" mais crítico e não está bloqueada por um "rendezvous". Se t pertence a uma região proibida então o processador permanece desocupado até o final dessa região.

O teorema 5 coaduna os resultados obtidos acima, formalizando a estratégia proposta por Mok.

Teorema 5. Se existe um escalonamento factível para um conjunto de tarefas com as primitivas de *rendezvous e monitor,* então o escalonamento com o modelo "kernelized monitor" produz um escalonamento válido.

No próximo capítulo são apresentadas três implementações de escalonamentos para sistemas monoprocessadores. O objetivo dessas implementações é sedimentar os conhecimentos obtidos no estudo dos algoritmos de escalonamento para sistemas "Hard Real Time".

4. IMPLEMENTAÇÃO DE UM NÚCLEO "HARD REAL TIME"

Com o objetivo de analisar alguns algoritmos de escalonamento, associado à necessidade existente no Centro Tecnológico para Informática de um núcleo de tempo real capaz de atender a projetos existentes na área de automação, foi formalizada a especificação de um núcleo que possui como objetivos os seguintes requisitos.

1. Linguagem de Implementação de alto nível, de forma a facilitar o transporte para diversos ambientes.

2. Padronização da interface com o usuário.

3. Capaz de atender sistemas "Hard Real Time".

4. Facilidade de implementação de diferentes estratégias de escalonamento.

Perseguindo esses objetivos foi implementado um núcleo [AZE90b] com as seguintes características:

1. Implementado em linguagem C.

2. Utilização do padrão MOSI (Microprocessor Operating Systems Interfaces) [IEE87].

3. Sistemas uniprocessadores.

4. Tarefas independentes com respeito à ordem de precedência.

5. Escalonamento dinâmico e preemptivo, sendo implementadas as seguintes estratégias:

   - prioridade fixa,
   - "earliest deadline first",
   - "least laxity first".

6. Recursos ilimitados.

A escolha das estratégias de escalonamento citadas tem por base os elementos levantados no capítulo anterior para sistemas uniprocessadores.

É importante ressaltar que as técnicas EDF e LLF são ótimas, segundo Mok [MOK78], para tarefas independentes com preempção e tempos arbitrários de chegada. Quando se consideram outras classes de tarefas, segundo estudos de Locke et al [LOC75], as técnicas citadas são as políticas heurísticas que apresentam melhores resultados.

A técnica com prioridade fixa é interessante sob o aspecto de comparação com as demais, além de permitir implementações de estratégias estáticas como a Taxa Monotônica [LIU73].

Nos próximos itens serão descritas as funções da norma MOSI implementadas, as estruturas de dados utilizadas, bem como detalhes das estratégias de escalonamento.

4.1. Funções do Núcleo

O objetivo da norma MOSI é fornecer um conjunto de convenções de interface para aumentar a portabilidade de programas de aplicação, no seu acesso aos serviços do sistema, quando eles são transportados através de diversos ambientes operacionais [MOO86].

As funções definidas pelo padrão são agrupadas de acordo com seu propósito em categorias denominadas "capabilities".

Cada "capability" é formada de um ou mais módulos, denominados "capability modules", que por sua vez são formados por um conjunto de funções logicamente relacionadas por um grupo de recursos sobre os quais elas operam.

O objetivo das "capabilities" definidas pelo padrão é atender todas as funções existentes nos diversos ambientes. As "capabilities" definidas são listadas abaixo:

- "Memory Management"
- "Time Management"
- "Data Transfer"
- "Data Management"
- "Process Management"
- "Process Synchronization and Communication"
- "Interface with the Environment"
- "Exception Handling"

Certas coleções de "capabilities" são recomendadas pelo padrão para suportar classes específicas de aplicações. Essas configurações permitem uma implementação ser aderente à norma, apesar de não atender todas as aplicações.

De acordo com nossos objetivos, listados no item anterior, a configuração implementada consiste das seguintes "capabilities":

- "Memory Management"
- "Time Management"
- "Process Management"
- "Process Synchronization and Communication"
- "Exception Handling"

Além destas "capabilities" foi implementada uma extensão à norma, para atender a interface de Aplicação do Protocolo MAP 3.0 denominada "Event Management".

A Estrutura de software do núcleo pode ser observada na figura 4.1..

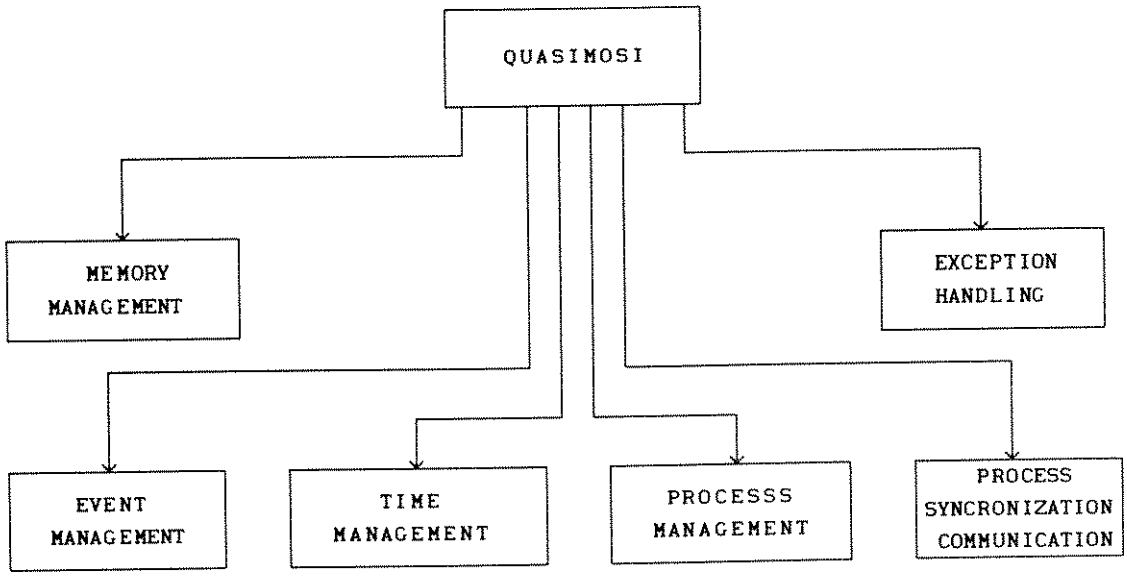

Figura 4.1. Estrutura do Software.

A seguir serão descritos os objetivos de cada "capability" implementada, bem como as funções associadas. Ao final de cada descrição são apresentadas as funções que compõem cada "Capability Module".

A descrição realizada é extremamente sucinta; caso o leitor deseje uma descrição mais profunda deve recorrer aos manuais do usuário, onde cada função é descrita em detalhes [AZE89a, AZE89b].

4.1.1. "Memory Management"

Esta "capability" descreve a interface do gerenciador de memória do executivo, responsável pela aquisição e liberação de blocos de memória livres existentes no sistema.

Como pode ser observado na figura 4.2. esta "capability" é composta por dois módulos: "Simple Memory" e "Multispace Memory".

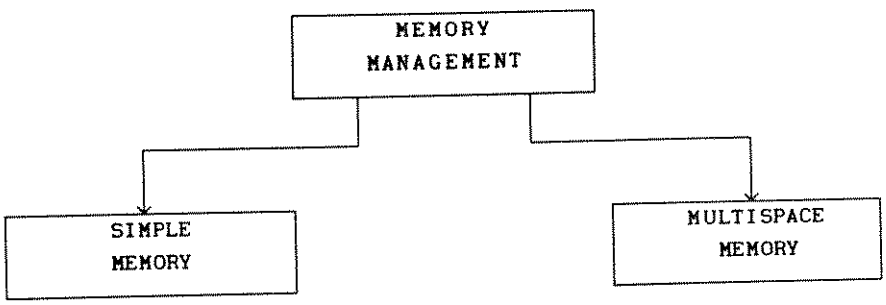

Figura 4.2. "Memory Management".

Os blocos de memória são ordenados pelo endereço crescente, sendo a alocação de blocos realizada através do algoritmo "first fit" elaborado por Knuth [KNU73].

Esta estratégia não é definitiva, algoritmos alternativos como "best fit" [KNU73] podem ser considerados dependendo do grau de utilização da memória e do tempo de resposta desejado.

Fazem parte desta "capability" as seguintes funções:

- "Simple Memory"
  ALLOCATE MEMORY. Aloca um bloco de memória.
  FREE MEMORY. Libera bloco previamente alocado.
  GET ALLOCATION UNIT. Fornece tamanho da unidade de alocação.
  GET AMOUNT REMAINING. Indica quantidade de memória disponível.
  GET BLOCK SIZE. Fornece tamanho do bloco adquirido.

- "Multispace Memory Management"

  MAKE POOL. Cria um "pool" de memória.

  FREE POOL. Libera "pool" de memória previamente alocado.

  ALLOCATE FROM POOL. Aloca área do "pool" especificado.

  FREE FROM POOL. Libera área previamente alocada.

4.1.2. "Time Management"

Esta "capability" tem por objetivo fornecer um método padronizado para programas de aplicação utilizarem temporizadores do sistema.

Dessa forma, este módulo é, dentre os módulos existentes na norma MOSI, aquele que possui o maior grau de relacionamento com o sistema hospedeiro.

Para evitar que o núcleo seja sobrecarregado com cálculos intermediários para a obtenção de frações de tempo (minuto, dia, hora, etc), é altamente recomendável que toda a manipulação com temporizadores seja realizada através de hardware próprio.

A figura 4.3. apresenta o relacionamento dos módulos que compõem a "capability".

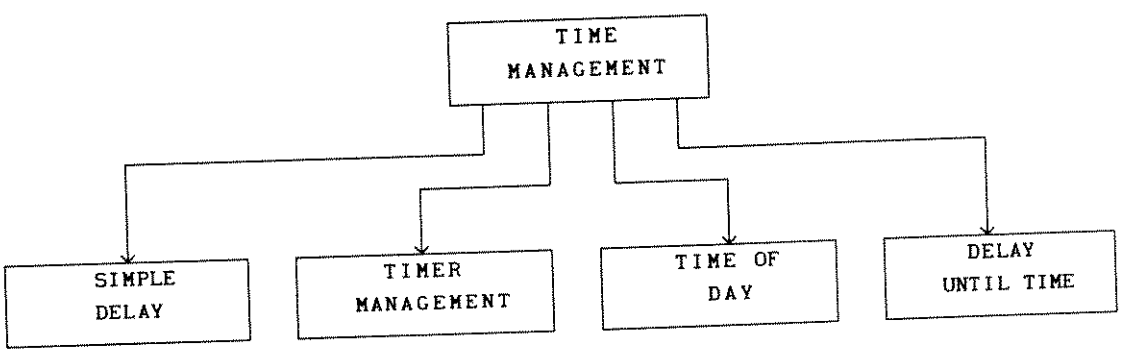

Figura 4.3. "Time Management".

É interessante analisar o algoritmo utilizado na implementação do módulo "Simple Delay", responsável pelo controle das temporizações do sistema.

O núcleo deve acompanhar, a cada pulso do relógio do sistema, a evolução das tarefas que estão em temporização.

A variável "osleep" mantém o ponteiro para o ínicio da lista de tarefas em temporização.

As tarefas na lista "osleep" são ordenadas através do tempo no qual sua temporização se encerrará; cada entrada na lista informa o número de "clock ticks" que a tarefa ainda possui além da entrada precedente na lista.

A primeira tarefa em "osleep" possui o menor tempo, a segunda possui como tempo total a soma de seu tempo e de sua antecessora, e assim sucessivamente até a última.

Esta organização permite que a atualização dos tempos seja realizada com operações somente sobre o primeiro elemento da lista, otimizando o processamento da interrupção.

Para facilitar ainda mais, o ponteiro "ostimtop" aponta diretamente para este valor. A lista de tarefas em temporização é denominada "delta lista" [COM85].

Fazem parte desta "capability" as seguintes funções:

- "Simple Delay"
  DELAY. Suspende uma tarefa por um tempo determinado.

- "Timer Management"
  CONNECT TIMER. Associa um temporizador à tarefa.
  DISCONNECT TIMER. Libera temporizador previamente associado.
  INITIALIZE TIMER. Inicializa o temporizador.
  CONTROL TIMER. Controla a operação do temporizador.
  READ TIMER. Determina o valor corrente do temporizador.
  GET EXPIRED TIMER. Identifica qual temporizador encerrou contagem.

- "Time of Day"
  GET TIME. Obtém a hora corrente.
  GET DATE. Obtém a data e a hora corrente.

- "Delay Until Time"
  SLEEP UNTIL. Suspende a tarefa até a data/hora especificada.

4.1.3. "Process Management"

Esta "capability" define um conjunto de funções que suporta a criação e o controle de tarefas executadas concorrentemente.

Uma tarefa é a menor unidade que pode ser ativada para execução.

A tarefa inicial do usuário para o qual o sistema operacional entrega o controle é conhecida como "root process".

Para que o "root process" seja conhecido pelo núcleo ele deve chamar a função INITIALIZE_PROGRAM antes de ativar qualquer outra função definida pela norma.

Conforme apresentado na figura 4.4., esta "capability" possui tres módulos: "Single Process", "Multiple Process" e "Suspend and Resume".

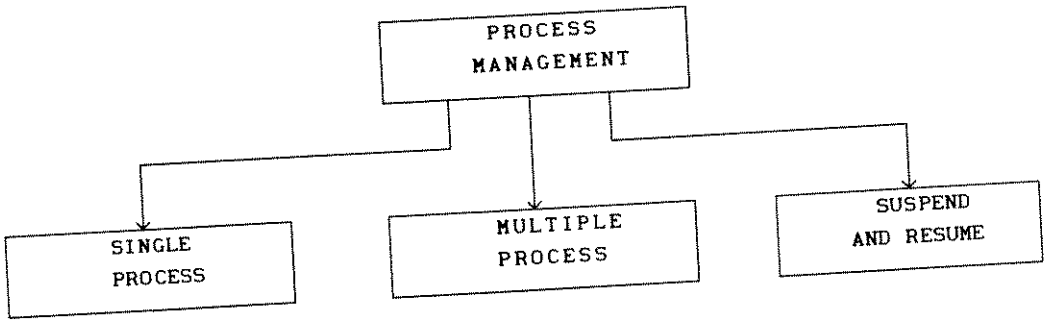

Figura 4.4. "Process management".

O módulo "Single Process" inicializa/termina a execução do núcleo além de implicitamente criar o "root process".

O módulo "Multiple Process" adiciona a funcionalidade requerida para criar e destruir tarefas descendentes, alterar prioridades e obter informações sobre estados das tarefas.

O módulo "Suspend and Resume" permite suspender e reiniciar uma tarefa.

A tabela II. apresenta os possíveis estados que uma tarefa pode assumir:

Tabela II. Tabela de estados.

TABELA DE ESTADOS	
ESTADO	SIGNIFICADO
PRCURR	em execução
PRREADY	pronto para executar
PRSUSP	suspenso
PRWAIT	aguardando semáforo
PRWT	aguardando/temporizando semáforo
PRFREE	acabou execução
PRSLEEP	dormindo
PRRECP	aguardando mensagem
PRRECT	aguardando/temporizando mensagem
PRRESP	aguardando resposta
PRREST	aguardando/temporizando resposta
PREVE	aguardando evento
PREVT	aguardando/temporizando evento

Compõem esta "capability" as seguintes funções:

- "Single Process"

  INITIALIZE PROGRAM. Habilita um programa a se inicializar.

  TERMINATE PROGRAM. Encerra a execução do programa.

- "Multiple Process"

  CREATE PROCESS. Cria uma tarefa.

  DESTROY PROCESS. Destrói uma tarefa.

  GET PROCESS STATUS. Determina o estado de uma tarefa.

  GET PROCESS INFO. Fornece à tarefa parâmetros passados na sua criação.

  CHANGE PRIORITY. Altera a prioridade de uma tarefa.

- "Suspend and Resume"

  SUSPEND PROCESS. Suspende uma tarefa.

  RESUME PROCESS. Reativa uma tarefa suspensa.

4.1.4. "Process Synchronization and Communication"

Esta "capability" fornece funções para viabilizar sincronização e comunicação num ambiente multi-tarefa.

A figura 4.5. descreve os diversos módulos que compõem esta "capability".

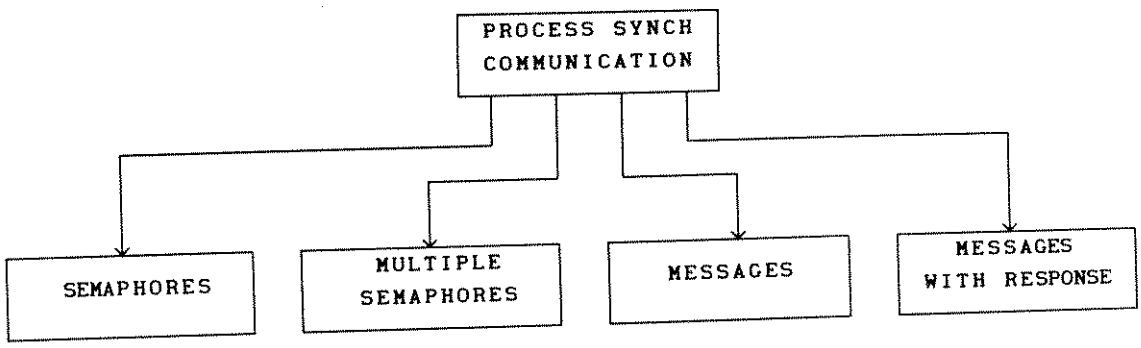

Figura 4.5. "Process Synchronization and Communication".

Semáforos são implementados para viabilizar sincronização entre tarefas.

Durante a inicialização o contador do semáforo recebe um valor inteiro positivo, sendo que o usuário recebe um identificador do semáforo que deverá ser utilizado nas próximas chamadas que envolvam o semáforo recém criado.

Quando uma tarefa sinaliza um semáforo e não existe nenhuma tarefa aguardando, o valor do semáforo é incrementado pela quantidade especificada.

No módulo "Semaphores" o valor do semáforo é incrementado sempre em valores unitários, seguindo a definição clássica de semáforos [HAN73].

No módulo "Multiple Semaphores" o valor do semáforo é incrementado de acordo com valores fornecidos pela tarefa, fato que representa uma alteração em relação à definição original, onde o acréscimo é sempre unitário. Essa característica não é encontrada usualmente nos núcleos existentes no mercado.

Se existirem tarefas aguardando e a sinalização liberá-las para execução, ocorrerá uma seleção de acordo com a política de escalonamento implementada.

Deve ser notado que, novamente, existe uma inovação em relação às implementações de núcleos convencionais. Normalmente, nessas implementações as tarefas aguardando uma sinalização são ordenadas por ordem de chegada, o que representa uma incoerência pois implica que a prioridade da tarefa não é reconhecida no acesso aos recursos.

Na nossa implementação a tarefa é suspensa para aguardar um recurso de acordo com a sua importância no sistema.

Quando uma tarefa aguarda por um semáforo e o contador do semáforo é menor que a quantidade requisitada, a tarefa será suspensa à espera de uma sinalização que a libere, ou até que uma temporização fornecida na chamada da função se encerre.

Os módulos "Messages" e "Messages with Response" são utilizados para comunicação entre tarefas através da troca de mensagens.

Esta comunicação é realizada através de "mailboxes", que servem como um ponto de troca para mensagens.

Tarefas que desejem enviar uma mensagem especificam a identificação da "mailbox" ao invés de um receptor específico.

A função SEND_MESSAGE não faz com que a tarefa geradora da mensagem fique bloqueada; a mensagem é copiada para uma área interna na mailbox e a tarefa continua sua execução normalmente.

Caso ocorra falta de recursos a tarefa é informada através de um código de erro.

Tarefas que desejem receber uma mensagem especificam uma "mailbox", ao invés de uma tarefa geradora da mensagem.

Se não existe mensagem disponível a tarefa deverá aguardar até um tempo limite especificado pelo usuário.

Caso exista a necessidade de respostas, um módulo específico denominado "Messages with Responses" deve ser utilizado para coordenar o envio/recepção de respostas.

Nesse caso, a dinâmica da troca de mensagens é acrescida de alguns elementos. Para ilustrar a troca de mensagens com respostas vamos utilizar um exemplo onde duas tarefas (T_1, T_2) realizam comunicação.

Suponha que a tarefa T1 envia uma mensagem para a tarefa T2; após a chamada da função "SEND MESSAGE" a tarefa T1 recebe um identificador que representa a transação dentro do sistema.

Caso exista a necessidade de resposta a tarefa geradora da mensagem (T_1) deve aguardar a resposta através da função "WAIT RESPONSE". Naturalmente, a resposta é gerada pela tarefa que recebeu a mensagem enviada (T_2), operação que é realizada através da função "RESPOND MESSAGE" (figura 4.6.). É necessário salientar que a resposta está associada à transação existente no sistema e não à tarefa T_1.

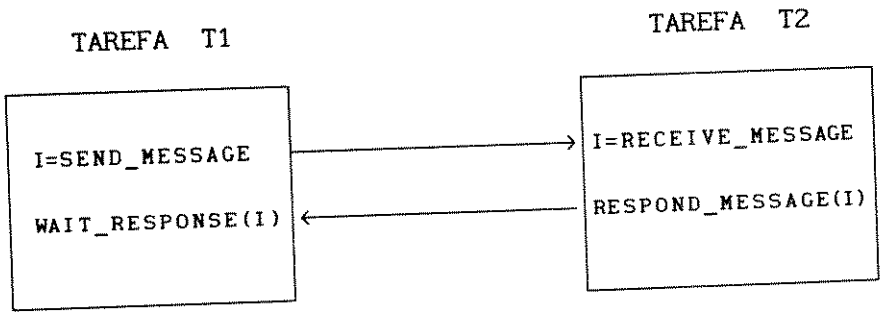

Figura 4.6. Mecanismo de Mensagem com Resposta.

Um ponto importante relacionado com a comunicação por meio de "mailboxes" é a forma como a memória é alocada para o armazenamento das mensagens.

A técnica implementada consiste em alocar memória do pool principal do sistema.

Esta solução pode conduzir a um "deadlock" visto que uma única tarefa pode consumir toda a memória existente.

Soluções alternativas podem ser consideradas como pode ser observado na tabela III.

Tabela III. Soluções alternativas de gerência de memória.

SOLUÇÃO	ALTERAÇÃO
aloca de um pool	Norma MOSI p/ na criação de uma mailbox identificar o POOL associado
aloca de um buffer	Norma MOSI p/ na criação de uma mailbox indicar o tamanho do buffer a ser associado a mail - box
limita nro de msg	Norma MOSI p/ na criação de uma mailbox indicar o nro máximo de mensagens permitidas

As funções que compõem esta "capability" são descritas a seguir:

- "Semaphores"

  ALLOCATE SEMAPHORE. Aloca um semáforo.

  SIGNAL SEMAPHORE. Sinaliza semáforo.

  WAIT SEMAPHORE. Aguarda sinalização num semáforo.

  DEALLOCATE SEMAPHORE. Libera semáforo previamente alocado.

- "Multiple Semaphores"

  MULTIPLE SIGNAL SEMAPHORE. Sinaliza semáforo com mais de um evento.

  MULTIPLE WAIT SEMAPHORE. Aguarda mais de um evento num semáforo.

- "Messages"

  ALLOCATE MAILBOX. Aloca "mailbox".

  SEND MESSAGE. Envia message.

  RECEIVE MESSAGE. Recebe mensagem.

  DEALLOCATE MAILBOX. Libera "mailbox" previamente alocada.

- "Messages with Responses"

  RESPOND MESSAGE. Responde mensagem.

  WAIT RESPONSE. Aguarda resposta.

4.1.5. "Exception Handling"

Esta "capability" define funções que habilitam o tratamento de exceções.

O termo exceção é utilizado para descrever um evento exigindo tratamento que interrompa o ciclo normal de execução.

Desta forma, exceções podem resultar de uma falha de hardware, interrupções de hardware (timer, I/O), erros de programa, interrupções de software (TRAPs), etc.

Os módulos que compõem esta "capability" podem ser observados na figura 4.7..

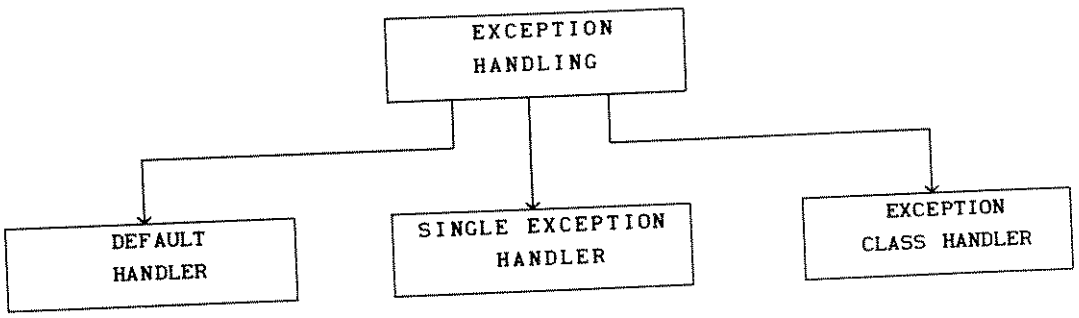

Figura 4.7. "Exception Handling".

Exceções são identificadas por uma classe e por uma subclasse, como apresentado na tabela IV..

Tabela IV. Classes de exceções.

	CLASS	SUBCLASS	EXCEPTION
Procedural Exception	1 1 1 1 1	1 2 3 4 5	Divide by zero Bounds error Overflow Underflow Stack overflow
Operador Intervention	2	-	
Timer	3	Identificador do timer	
Vetores de Interrupções	4 . .	- . -	

Cada uma das exceções reconhecidas pelo Executivo pode ser manuseada por um "system default handler" ou um "user-defined handler".

O "system default handler" atuará sobre cada exceção do modo descrito pela tabela V..

Tabela V. Formas de atuação do "default handler".

	CLASS	ATUAÇÃO
Procedural Exception	1	Destrói o tarefa que causou a exceção, todos os recursos serão liberados.
Operador Intervention	2	Ignora
Timer	3	Ignora
Vetores de Interrupção	4 . .	Ignora

Uma tarefa herda o "handler exception" de seu criador. O "default handler" é mantido a menos que seja alterado através da função SET_EXCEPTION_HANDLER.

Para cada classe de exceção, uma tarefa pode habilitar ou desabilitar o tratamento de exceções; se uma classe é desabilitada, as exceções desta classe são efetivamente ignoradas.

O estado inicial dos "handlers" de exceção é desabilitado; desta forma, se o usuário desejar utilizar o handler ele deve habilitar a classe associada através da função ENABLE EXCEPTION CLASS.

É evidente que o dispositivo gerador das exceções só pode ser ativado após as inicializações descritas acima, pois caso contrário pode ocorrer perda de exceções.

Quando um "exception handler" está ativo, é assumido que nenhuma outra exceção pode ser ativada até ele encerrar o processamento.

Fazem parte desta "capability" as seguintes funções:

- "Default Handler"
  DISABLE EXCEPTION CLASS. Inibe o tratamento de exceções da classe requerida.
  ENABLE EXCEPTION CLASS. Habilita o tratamento de exceções da classe requerida.
  GET ENABLE STATUS. Retorna o estado da classe requerida.

- "Single Exception Handler"
  GET EXCEPTION HANDLER. Obtém o "handler" corrente.
  SET EXCEPTION HANDLER. Altera o "handler" corrente.
  RESET DEFAULT HANDLER. Retorna ao "handler default".
  RAISE EXCEPTION. Gera uma exceção por software.
  GET EXCEPTION CODE. Identifica a exceção que ocorreu.
  EXIT FROM HANDLER. Encerra a execução do handler.

- "Exception Class Handlers"
  GET EXCEPTION CLASS HANDLER. Obtém o "handler" associado à classe.
  SET EXCEPTION CLASS HANDLER. Associa um novo "handler" para a classe especificada.

4.1.6. "Event Management"

Esta "capability" introduz uma extensão à norma MOSI, para atender a Interface de Aplicação do Protocolo MAP 3.0.

Basicamente ela implementa semáforos clássicos com a possibilidade de uma tarefa aguardar a ocorrência de um conjunto de eventos.

Desta forma, o núcleo permite que a tarefa defina qual evento ou combinação de eventos devem ser aguardados e, implicitamente, remover o evento no momento em que a tarefa é notificada da sua ocorrência.

A figura 4.8. descreve os módulos que compõem esta "capability".

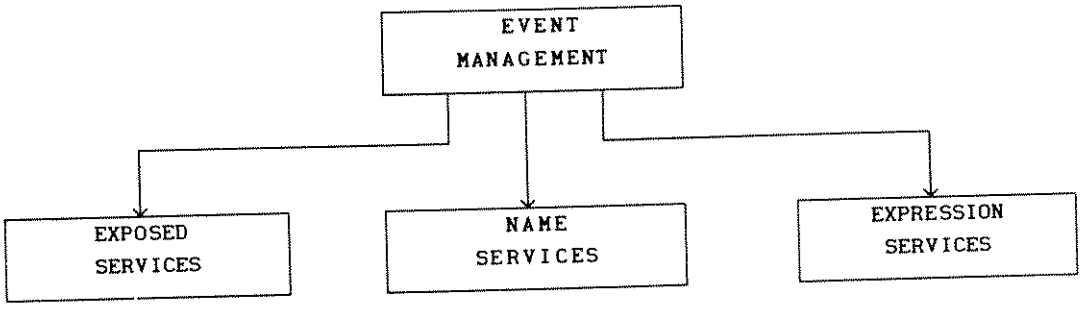

Figura 4.8. "Event Management".

Compõem esta "capability" as seguintes funções:

GET EVENT NAME. Define um evento.

DELETE EVENT. Libera o evento.

EXPRESSION EVENT ADD. Adiciona o evento à expressão de eventos da tarefa.

EXPRESSION EVENT DELETE. Remove o evento da expressão de eventos da tarefa.

WAIT EVENT. Aguarda a sinalização de um evento.

NOTE. Sinaliza um evento.

4.2. O configurador do núcleo

As aplicações às quais o núcleo se destina são as mais diversas possíveis, possuindo características particulares que modelam o sistema tanto em tamanho quanto em capacidade.

Nesse contexto, o sistema operacional deve ser orientado para a aplicação, de forma a evitar que devido à existência de funções desnecessárias, seja gerada uma configuração maior do que a necessária.

A geração automática do núcleo de tempo real direcionada para a aplicação resolve o problema acima.

Devido à implementação modular do núcleo, é possível obter configurações apropriadas a cada aplicação, permitindo, inclusive, a alteração de estratégias internas ao núcleo como, por exemplo, o salvamento dos registradores do coprocessador.

Apesar do configurador influenciar estruturas internas do núcleo, o operador não necessita nenhum conhecimento desses elementos, sendo toda a interface realizada através de diálogos com o programa gerador.

A figura 4.9. ilustra a sequência utilizada na geração do núcleo.

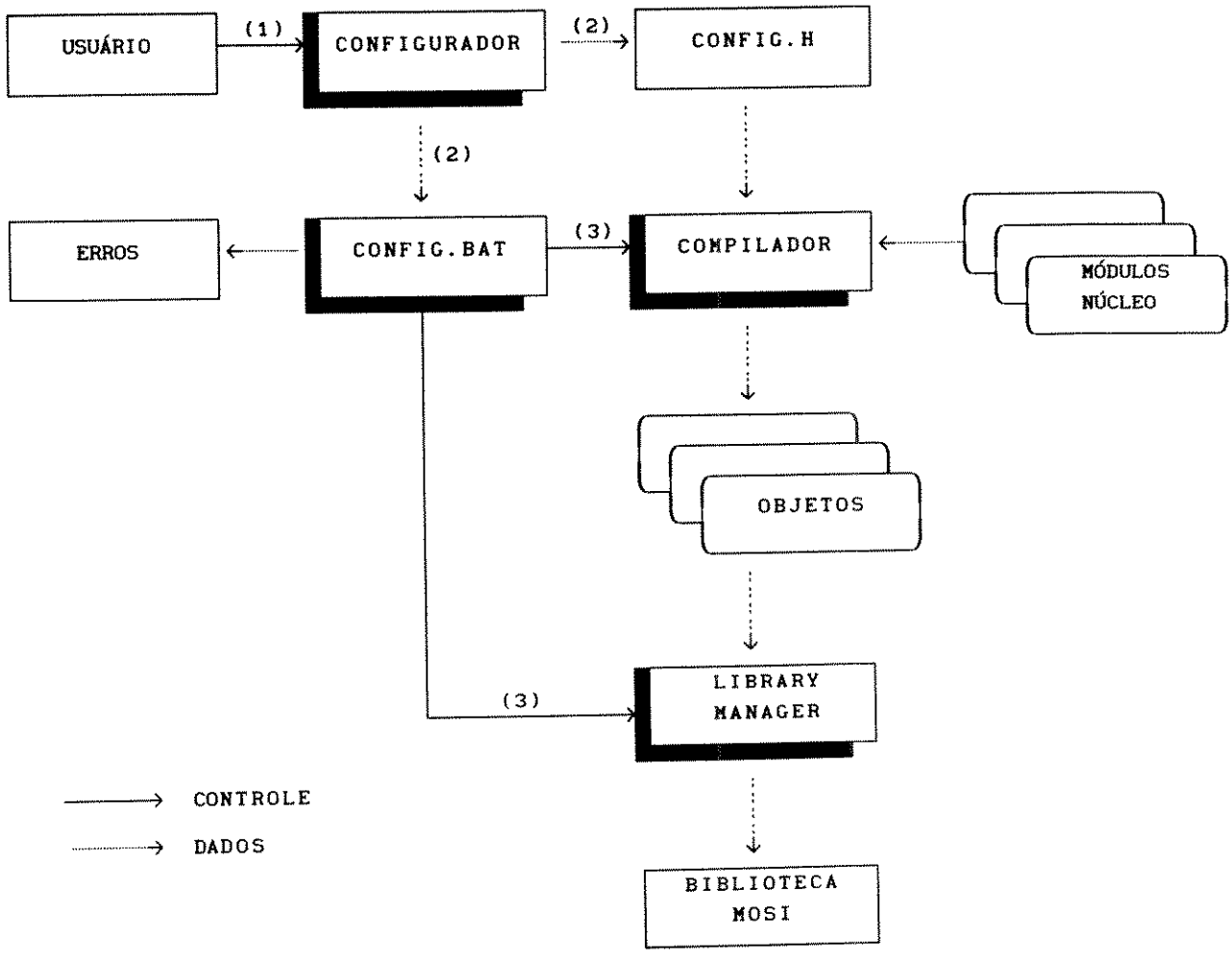

Figura 4.9. Dinâmica da Geração do Núcleo.

O usuário fornece como entrada para o configurador a especificação do seu sistema (1); após encerrar todo o diálogo com o usuário o configurador gera dois arquivos: config.h e config.bat (2). O arquivo config.h contém constantes e indicadores que serão incluídos nos arquivos fontes direcionando a geração. O arquivo config.bat determina a sequência de compilação e "Library Manager" necessária para atender ao usuário (3).

Finalmente, como resultado de todo esse processo é obtida uma biblioteca que será utilizada pelo programa de aplicação na implementação definitiva.

As opções de configuração oferecidas ao usuário são especificadas nas classes descritas a seguir:

a) Direcionar as "capabilities" presentes no núcleo para a aplicação, selecionando uma combinação dos seguintes elementos:
   - semáforos,
   - "mailboxes",
   - eventos,
   - "pools" de memória,
   - tratamento de exceções.

b) Limitar a quantidade de memória utilizada através do controle dos seguintes recursos:
   - número de processos,
   - número de semáforos,
   - número de "mailboxes",
   - número de eventos,
   - número de "pools" de memória,
   - número de temporizadores,
   - número de exceções.

c) Controlar a integração do núcleo com o ambiente através das seguintes características:
   - salvamento ou não dos registros do coprocessador,
   - seleção do período dos "ticks" do sistema,
   - associação dos vetores de interrupção aos tratadores de interrupção.

d) Associar rotinas do usuário com o tratamento de situações específicas criadas na utilização do núcleo, como por exemplo:
   - tratamento de "estouro de deadline" da tarefa,
   - inicialização de "hardware".

Naturalmente, muitas outras características poderiam ser alvo de atenção como, por exemplo: algoritmos específicos para alocação de memória ou tratamento de filas. Entretanto, apesar de suas limitações o configurador tem se mostrado como uma poderosa ferramenta na geração de sistemas apropriados às necessidades do usuário.

Como forma de ilustrar a utilização do configurador as telas geradas durante a sua execução são apresentadas no anexo 2.

4.3. Estrutura de dados

As estruturas de dados utilizadas para representar as informações presentes no núcleo são formadas basicamente por listas encadeadas associadas a algoritmos de inserção e retirada de elementos.

Como forma de organizar esta descrição as estruturas serão agrupadas nas "capabilities" às quais estão associadas.

4.3.1. "Process Management"

Do ponto de vista do software esta "capability" mantém as estruturas mais importantes do núcleo.

Nela estão representados os elementos que descrevem as tarefas, além das filas associadas aos estados do núcleo.

O descritor de tarefas é responsável pelo controle de todas as atividades relacionadas a uma tarefa. Ele contém os campos descritos na tabela VI..

Tabela VI. Tabela de estados.

ESTRUTURA DE PROCESSOS	
NOME	DESCRIÇÃO
pprox	aponta para o próximo descritor
pstate	estado da tarefa
pdesc	descendência do proc: RAIZ ou FILHO
pret	razão do término da execução
pregs	registradores da tarefa
psize	tamanho da área alocada pela tarefa
plimit	limite inferior da pilha
pargs	parâmetros recebidos pela tarefa
pnargs	tamanho dos parâmetros recebidos
pfila	aponta p/ início da lista onde task está
ptimer	mantém o valor de temporizações
pptim	aponta próx tarefa na lista de temporiz
psemcnt	contador para "multiple semaphores"
pmsg	aponta para buffer contendo mensagem
plmsg	fornece tamanho da mensagem
pidmsg	identificador da mensagem

Os descritores de tarefas podem estar associados a diversas listas, sendo que merecem destaque as apresentadas na tabela VII..

Tabela VII. Listas de processos.

NOME	DESCRIÇÃO
rdy	mantém as tarefas prontas p/ executar
currpid	mantém a tarefa em execução
osleep	mantém as tarefas em temporização

Um dos pontos mais importantes associados à gerência de processos está associado à inserção de tarefas na lista "rdy".

Dependendo do algoritmo de escalonamento escolhido, essa lista pode ser definida como possuindo dois níveis: um para tarefas com "deadline", e outro para tarefas com prioridade fixa.

Essa organização, bem como os algoritmos utilizados para sua manipulação, serão descritos no item que trata dos algoritmos de escalonamento.

4.3.2. "Process Synchronization and Communication"

Naturalmente nesta "capability" estão os descritores dos semáforos e das mailboxes.

Vale enfatizar que as tarefas aguardando uma sinalização ou uma mensagem são organizadas de acordo com o tipo de escalonamento implementado.

Dessa forma os algoritmos de inserção e retirada serão os mesmos utilizados na lista "rdy".

O descritor de semáforo deve manter o valor do contador associado ao semáforo, bem como a lista das tarefas que estejam aguardando uma sinalização.

Esse descritor contém os campos descritos na tabela VIII..

Tabela VIII. Campos do descritor do semáforo.

NOME	DESCRIÇÃO
sstate	estado do semáforo (livre/ocupado)
semcnt	contador do semáforo
sqhead	aponta para o início da fila de task

O descritor de mailbox mantém, além da lista das tarefas aguardando uma mensagem, uma fila com as mensagens recebidas; seus campos podem ser observados na tabela IX..

Tabela IX. Campos do descritor de mailbox.

NOME	DESCRIÇÃO
mstate	estado da mailbox (livre/ocupada)
phead	aponta p/ início da lista de tarefas
msghead	aponta p/ início da fila de mensagens
msgtail	aponta p/ final de fila de mensagens

Para implementar o mecanismo de mensagens com respostas existe uma variável denominada "respostas" que contém apontadores para as tarefas aguardando respostas e para respostas que já chegaram.

Como as respostas são direcionadas para cada tarefa, o mecanismo de inserção de tarefas é diferente do utilizado nas estruturas anteriores.

Nesse caso as tarefas são organizadas numa fila, que é pesquisada linearmente a cada resposta recebida. Algoritmos alternativos de pesquisa poderiam ser utilizados; no entanto, seu custo de utilização somente seria viável num sistema com grande volume de mensagens com respostas, o que não ocorre com os requisitos desta implementação.

4.3.3. "Memory Management"

A gerência de memória é organizada através de uma lista encadeada ligando cada bloco de memória disponível.

Dessa forma, cada bloco deve conter informações a respeito de seu tamanho, bem como de seu sucessor.

A tabela X. descreve os campos que compõe essa estrutura.

Tabela X. Campos do descritor de memória.

NOME	DESCRIÇÃO
mnext	aponta para próximo bloco
mlen	tamanho da área

Os blocos são organizados por endereço, de modo que os blocos iniciais apontam para áreas de memória com endereçamento baixo, que crescem à medida que se avança para os blocos finais.

A figura 4.10. fornece uma noção dessa organização.

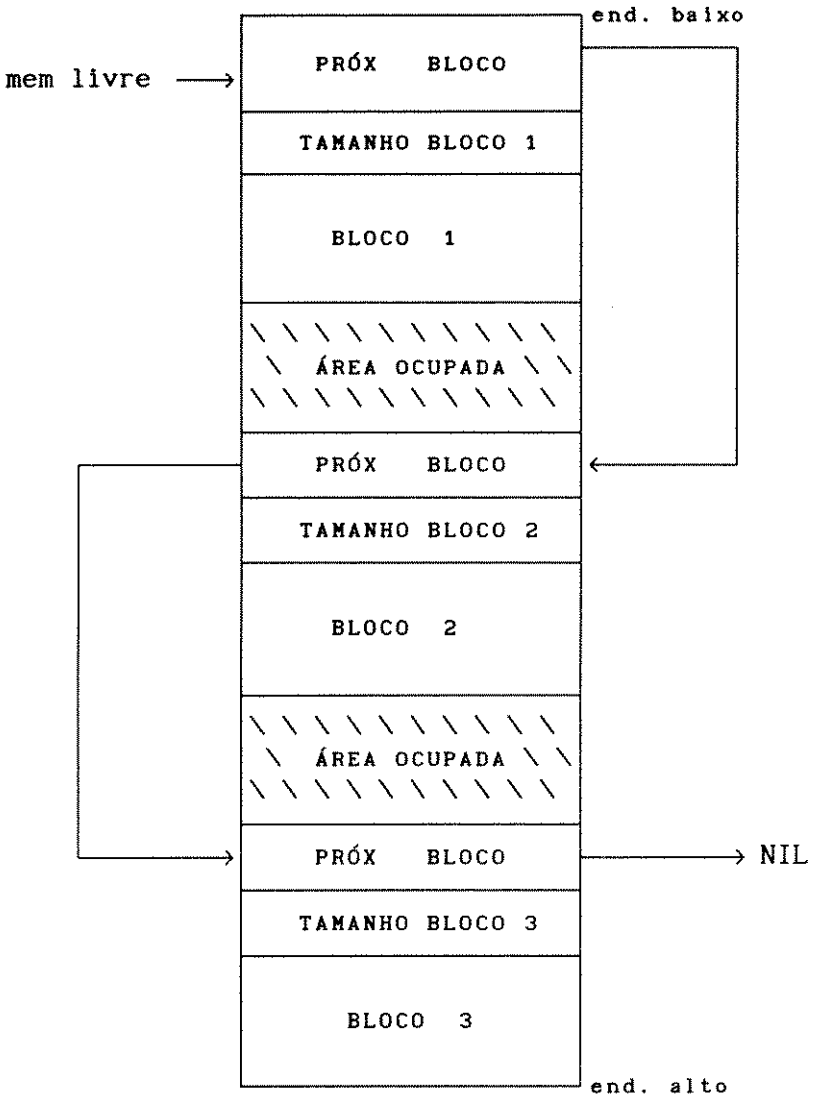

Figura 4.10. Mapa de memória do núcleo.

Quando ocorre uma liberação de memória, é inicialmente tentada uma associação com blocos adjacentes antes de se criar um novo bloco e sua inserção na lista de blocos livres.

4.3.4. "Exception Handling"

Para controlar exceções é necessário manter uma estrutura que contenha as rotinas responsáveis pelo tratamento das interrupções associadas a cada classe definida pelo usuário.

Para realizar esse controle existe uma estrutura ("interrupt") cujas entradas contém os campos descritos na tabela XI..

Tabela XI. Campos do descritor de exceções.

NOME	DESCRIÇÃO
estate	estado da entrada (livre/ocupada)
hdlclass	classe da exceção
funcint	aponta p/ handler do usuário

4.3.5. "Event Management"

O controle de múltiplos eventos exige estruturas mais complexas que as utilizadas no controle de semáforos.

Inicialmente é necessário agrupar os eventos numa única tabela, denominada "osevent".

As entradas dessa estrutura possuem os campos descritos na tabela XII.

Tabela XII. Campos do descritor de eventos.

NOME	DESCRIÇÃO
evstate	estado da entrada (livre/ocupada)
evcont	contador de eventos
evproc	número de tarefas associadas

Além dessa tabela é necessário manter associada a cada tarefa uma descrição dos eventos aos quais a tarefa está associada.

Essa estrutura é denominada "ospet", onde cada entrada possui os campos descritos na tabela XIII..

Tabela XIII. Campos de identificação de eventos.

NOME	DESCRIÇÃO
expr	expressão de eventos
pexp	índice para próxima entrada na expressão
event	identifica evento sinalizado

4.4. Estratégias de Escalonamento

Como já afirmado anteriormente foram implementadas três estratégias distintas de escalonamento [AZE90a].

A primeira consiste do clássico sistema de escalonamento por prioridade fixa. Esse escalonamento pode ser utilizado com estratégias estáticas como a "Rate Monotonic" [LIU73] e suas extensões para atender tarefas aperiódicas [SPR89]. É necessário ressaltar a implementação de uma extensão ao núcleo básico que permite utilizar o protocolo Limite de Prioridade no acesso às regiões críticas do sistema.

O trabalho de Mok e Dertouzos [MOK78] afirma que, para tarefas independentes em sistemas uniprocessadores, as estratégias dinâmicas "earliest deadline first" e "least laxity first" são ótimas e podem ser aplicadas em sistemas "Hard Real Time".

Dessa forma, as duas outras estratégias implementadas são exatamente as citadas no trabalho acima, com algumas alterações para manipular tarefas que não possuem "deadlines".

Essas alterações foram realizadas pelo fato desses algoritmos não levarem em consideração o tratamento a ser dado a tarefas que não possuam "deadlines" como, por exemplo: estatística, relatórios, etc.

Uma forma de tratar a questão é alocar "deadlines" infinitos para tais tarefas; entretanto, tal estratégia coloca todas em um mesmo nível de "prioridade", o que em algumas situações não é desejável.

A técnica utilizada na implementação é similar ao escalonamento por "feedback queues" (capítulo 4), no que diz respeito à existência de diversos níveis nas filas do escalonador. No nosso caso, as filas do núcleo passaram a possuir dois níveis, o primeiro associado às tarefas que possuem "deadlines" e o segundo associado às tarefas com prioridade.

É importante salientar que a existência de dois níveis não compromete o fato dos algoritmos "earliest deadline first" e "least laxity first" serem ótimos, uma vez que as tarefas com prioridade somente estarão em execução

quando não existirem tarefas no primeiro nível, aproveitando o tempo "ocioso" do processador.

Este tratamento constitui uma inovação nos algoritmos de escalonamento para tempo real e facilita a especificação do sistema criando uma classe para tarefas que não possuem "deadlines".

Nos próximos ítens as estratégias citadas serão descritas, além de um exemplo comparativo entre as mesmas.

4.4.1. Escalonamento com Prioridade Fixa - QUASIMOSI

Nesta estratégia cada tarefa, na sua criação, recebe um valor inteiro positivo que representa a sua prioridade dentro do sistema.

Essa prioridade não é alterada dinamicamente pelo núcleo, permanecendo constante até o fim da execução, salvo mudanças realizadas pelo próprio aplicativo através da função CHANGE_ PRIORITY.

Nesta estratégia a tarefa mais prioritária em condições de executar sempre estará ocupando a CPU, que somente será liberada após o término de sua execução, numa liberação espontânea realizada pela tarefa ou no caso de uma tarefa mais prioritária atingir o estado de pronto.

Um instantâneo das filas que mantêm as tarefas em execução (currpid) e as tarefas prontas para executar (rdy), com escalonamento por prioridade fixa, pode ser observado na figura 4.11..

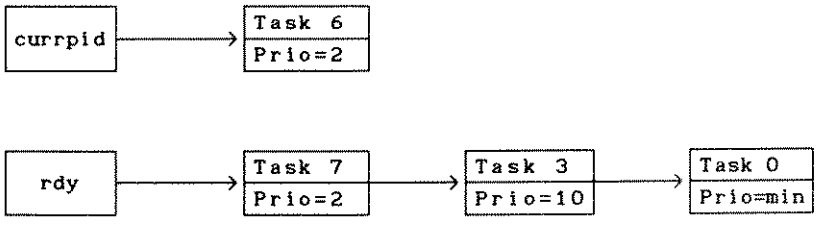

Figura 4.11. Filas com prioridade fixa.

A tarefa 0 possui a menor importância no sistema, pois sua única função é manter as estruturas coerentes quando não existem tarefas prontas para executar.

Dessa forma, o sistema não permanece executando ao nível do núcleo quando não existem mais tarefas do usuário na fila de prontos. Nesse caso é ativada a tarefa 0 cujo único objetivo é retornar ao nível do usuário onde interrupções são habilitadas permitindo a mudança de estado de tarefas que estejam temporizando ou aguardando recursos.

O corpo da tarefa 0 consiste de um "loop" infinito contendo em seu interior uma instrução de "HALT" que libera o barramento para outras atividades do sistema.

O núcleo resultante desta implementação foi denominado QUASIMOSI.

O núcleo QUASIMOSI pode ser utilizado em sistemas "Hard Real Time" se considerarmos a estratégia Taxa Monotônica descrita pelo item 3.7.1..

O algoritmo Taxa Monotônica atribui prioridades fixas às tarefas do sistema de acordo com seu período. Ele é apropriado para tarefas periódicas, preemptíveis e independentes.

A restrição de independência pode inviabilizar a aplicação do algoritmo em sistemas onde a sincronização e comunicação entre as tarefas sejam necessárias.

Para contornar essa deficiência foi implementada uma extensão ao núcleo QUASIMOSI para atender ao Protocolo Limite de Prioridade [SHA90b] descrito no item 3.7.4..

Os principais pontos alterados para a implementação do protocolo são descritos a seguir:

a) Na função ALLOCATE SEMAPHORE (aloca um semáforo) foi acrescentado um parâmetro para indicar o limite de prioridade do semáforo em questão.

b) Na estrutura que representa o semáforo foram acrescentados dois novos campos:

"sempid" - identifica tarefa na região crítica.

"semlpr" - limite de prioridade do semáforo.

c) Foi definida uma fila de tarefas "oslimp" que mantém as tarefas bloqueadas pelo limite de prioridade.

d) O algoritmo que implementa a função WAIT SEMAPHORE (aguarda semáforo) foi alterado passando a possuir o seguinte formato:

```
liberado = FALSE.
Enquanto não liberado faça

  limite_prioridade = MINPRIO.          /* prioridade mínima */
  Para todos semáforos faça
      Se  semáforo_bloqueado e tarefa_bloqueante diferente da
          tarefa_corrente
      então Se limite_prioridade < limite_semáforo_bloqueado
            então limite_prioridade= limite_semáforo_bloqueado

  Se prioridade_processo_corrente > limite_prioridade
  então libera = TRUE.
  senão
      estado_processo_corrente = AGUARDANDO.
      insere processo_corrente na fila "oslimp".
      prioridade_próxima_tarefa_executar =
                                    prioridade_tarefa_corrente.
      ativa próximo processo.
```

Em resumo, o algoritmo acima efetua a seguinte operação:

Quando uma tarefa *J* pede para utilizar a região crítica controlada pelo semáforo *S*, inicialmente é verificado se sua prioridade é maior que o limite de prioridade de todos os semáforos correntemente bloqueados por tarefas distintas da tarefa *J*.

Se for maior, a tarefa *J* entra na região crítica.

Se for menor, a tarefa *J* será inserida na lista de tarefas bloqueadas e a próxima tarefa a ser ativada recebe a prioridade da tarefa *J*.

e) O algoritmo que implementa a função SIGNAL SEMAPHORE (sinaliza semáforo) foi também alterado passando a possuir o seguinte formato:

```
limite_prioridade = MINPRIO.          /* prioridade mínima */
Para todos semáforos faça
  Se semáforo bloqueado
  então Se limite_prioridade < limite_semáforo_bloqueado
        então limite_prioridade = limite_semáforo_bloqueado.

Para todas tarefas na fila "oslimp" faça
  Se prioridade_tarefa > limite_prioridade
  então insira tarefa na fila de PRONTO.

máxima_prioridade = MINPRIO.
Para todas tarefas na fila "oslimp" faça
  Se prioridade_tarefa > máxima_prioridade
  então máxima_prioridade = prioridade_tarefa.

Se prioridade_original_tarefa_corrente > máxima_prioridade
então prioridade_tarefa_corrente  =
                          prioridade_original_tarefa_corrente.
senão prioridade_tarefa_corrente = máxima_prioridade.

Re_escalonar_tarefas.
```

Em resumo, o algoritmo acima efetua a seguinte operação:

Todas as tarefas presentes na lista de tarefas bloqueadas cujas prioridades sejam maiores que o limite de prioridade de todos os semáforos correntemente bloqueados serão ativadas.

A tarefa "pronta para executar" recebe a maior prioridade presente na lista de tarefas bloqueadas, ou então, sua prioridade original, caso esta seja maior que a maior prioridade obtida na lista de tarefas bloqueadas.

É necessário ressaltar que este esquema é válido se não forem utilizadas temporizações e nenhum outro módulo de sincronização e comunicação entre tarefas. Além disso, sempre que for realizada uma operação *P(S)* sua operação dual *V(S)* também deve ocorrer.

4.4.2. Escalonamento "Earliest Deadline First" - DEADMOSI

Antes de descrever o algoritmo é necessário apresentar o elemento fundamental que será manipulado pelo mesmo: a tarefa.

O núcleo gerado nesta implementação foi denominado DEADMOSI.

a) *As tarefas do núcleo.* As tarefas que o núcleo reconhece estão divididas em três classes: periódicas com "deadline", não periódicas com "deadline", e não periódicas com prioridade.

   As tarefas periódicas com "deadline", são aquelas que ocorrem ciclicamente no sistema dentro de um intervalo de tempo previamente determinado, denominado período; ou seja, elas devem ser executadas uma vez a cada período, sendo que sua execução não pode "avançar" no período seguinte.

   As tarefas não periódicas com "deadline", são aquelas que ocorrem no sistema em tempos indeterminados.

   Estas tarefas (periódicas e não periódicas, com "deadline") possuem como característica um valor inteiro positivo denominado "deadline".

   O "deadline" representa o tempo D no qual a execução da tarefa deve ter sido encerrada, sob pena de prejudicar o funcionamento normal do sistema.

   As tarefas não periódicas com prioridade representam tarefas que não possuem um "deadline" na sua criação, ou seja, sua execução também deve ser eficiente, mas não existem limites críticos para o seu término; para controlar tais tarefas utilizaremos o clássico esquema de prioridades.

   Finalmente é necessário ressaltar que o único recurso manipulado pelo núcleo é a CPU, sendo todos os outros recursos considerados como ilimitados.

b) *O algoritmo.* O algoritmo de escalonamento está implementado em dois níveis: com "deadline" e sem "deadline".

No primeiro nível as tarefas com "deadline" são ordenadas em ordem crescente de "deadline"; dessa forma a tarefa no início da lista é a que possui o tempo para encerrar a execução próximo de seu limite, sendo então selecionada para execução. Uma vez que não existam mais tarefas no primeiro nível, o algoritmo continua a sua pesquisa no segundo nível, onde se encontram as tarefas ordenadas por prioridade.

Uma vez definidos os níveis é necessário alocar os tipos de tarefas definidas no ítem anterior ao nível apropriado. As tarefas não periódicas com prioridade serão alocadas no nível dois, de acordo com a prioridade definida na sua criação. As tarefas não periódicas com "deadline" serão naturalmente alocadas no nível um, de acordo com seus "deadlines". Resta somente alocar as tarefas periódicas com "deadline"; tais tarefas serão também alocadas no nível um, sendo seu "deadline" fornecido pelo usuário na sua criação.

c) *Características da implementação.* A estrutura modular do núcleo facilitou a implementação das alterações exigidas pelo novo algoritmo de escalonamento.

Nos próximos ítens são apresentadas as principais mudanças ocorridas.

- *A lista delta "osdead".* O núcleo deve acompanhar o progresso das tarefas que possuem "deadline" de modo a poder determinar se os limites de tempo são obedecidos garantindo a integridade do sistema.

  Este acompanhamento é realizado quando da interrupção associada ao relógio do sistema, onde as tarefas com "deadline" são verificadas para observar se houve ou não estouro do limite de execução.

  Para otimizar esta análise as tarefas serão organizadas numa lista denominada "delta lista", semelhante à utilizada para organizar as tarefas em temporização. A variável "osdead" mantém o ponteiro para o início da delta lista.

Tarefas na lista "osdead" são ordenados através do tempo no qual seu "deadline" se encerrará; cada entrada na lista informa o número de clock ticks que a tarefa ainda possui além da entrada precedente na lista.

A primeira tarefa em "osdead" possui o menor tempo, a segunda possui como tempo total a soma de seu tempo e de sua antecessora, e assim sucessivamente, até a última.

- *O descritor de tarefas (PID).* O descritor de tarefas é responsável pelo controle de toda atividade relacionada a uma tarefa. Dentro deste descritor foram inseridos e/ou alterados os seguintes campos:

ptipo - indica o tipo ao qual a tarefa está associada.

ppamtipo - indica o valor do parâmetro associado ao tipo.
De acordo com o tipo este parâmetro assume os valores descritos na tabela XIV..

Tabela XIV. Tipos de tarefas.

TIPO	PARÂMETRO
DEADLINE	deadline associado
PERIÓDICA	deadline associado
PRIORIDADE	prioridade associada

pdead - Aponta para o próximo elemento na lista osdead.

pchave - Mantém a chave de ordenação das tarefas do núcleo. No caso de tarefas com prioridade este campo contém a prioridade da tarefa; nos outros casos ele contém o tempo real de deadline, que é obtido através da soma do relógio do núcleo, no ínicio da tarefa ou no instante da ativação de uma tarefa periódica, com o deadline fornecido pelo usuário.

pdeltad - Representa o fator "delta" utilizado para ordenação na lista osdead.

pinicio - Ponteiro para início da tarefa.

pprxper- mantém o valor do relógio da próxima ativação de tarefas periódicas; este valor é utilizado na reinicialização da tarefa no período seguinte, simulando a função CREATE_PROCESS.

- *As listas do núcleo.* Nesta implementação do núcleo sempre que uma tarefa deve aguardar por um recurso ela é inserida numa lista associada ao recurso em questão.

  Esta inserção obedece à "importância" da tarefa no sistema; tal informação é fornecida pelo "deadline" ou pela prioridade.

  Entre os recursos gerenciados por listas temos: os semáforos, as "mailboxes" e o processador.

  Como exemplo de implementação utilizaremos a fila "rdy", que é responsável pelo controle das tarefas que necessitam do processador, ou seja, ela gerencia as tarefas que se encontram no estado de "pronto para rodar".

  Nesta implementação ela passou a possuir duas entradas, que estão associadas a cada um dos níveis descritos anteriormente, conforme pode ser observado na figura 4.12..

  Assim sua declaração passou a ser: *systag_t rdy[2].*

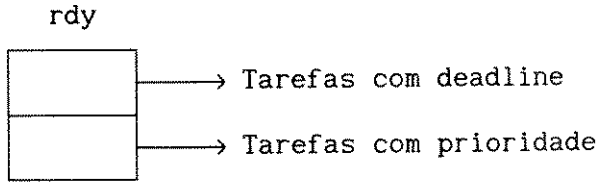

Figura 4.12. Estrutura das filas.

A inserção de tarefas com "deadline" obedece à sequência estabelecida através da entrada "pchave" presente no descritor de tarefas, ou seja, quanto menor for o valor de pchave, mais próximo do ínicio da lista a tarefa se encontra.

A inserção de tarefas com prioridade obedece à prioridade definida na sua criação.

- *A criação e destruição de tarefas.* Durante a criação de uma tarefa seu tipo é analisado e, caso ela pertença à classe de tarefas com "deadline", o ato de criação é acrescido de algumas ações que irão viabilizar sua implementação.

Uma das primeiras ações a serem executados é a inserção da tarefa na lista "osdead" através da rotina osinsdead().

Outra ação está relacionada com o cálculo do valor da entrada "pchave" presente na estrutura descritora de tarefas, que é realizado através da seguinte operação: pchave = ppamtipo + osclock.

Naturalmente, problemas podem surgir diante de um eventual estouro de "osclock". No entanto, se considerarmos o tick default do IBM/PC igual a 18.2 ticks/segundo ,a variável "osclock" deverá atingir o overflow após 65552 horas (ou 7.47 anos):

((0xffffffff/18.2)/60)/60= 65552horas = 7.47798 anos.

Todavia, como forma de prevenção, uma verificação de overflow é realizada após a atualização de "osclock" e também de "pchave", e no caso de overflow a rotina "osrecalculo()" recalculará todos os "pchave" das tarefas presentes na lista "osdead".

No caso de destruição da tarefa, seu tipo é analisado e, caso ela pertença à classe de tarefas com "deadline", ela deve ser retirada da lista "osdead" através da rotina "osretdead()".

Além disso, caso a tarefa seja periódica ela não é destruída no final de sua execução, mas sim "congelada" até o próximo período, quando então será reativada com as mesmas características de sua primeira ativação.

- *Estouro de "deadline"*. No caso de estouro de "deadline" será ativada a rotina "ostrdead()" que é responsável pelo tratamento do problema.

  Esta rotina é ativada do interior de uma rotina de tratamento de interrupção. Assim, no instante de sua chamada as interrupções estão desabilitadas.

  Basicamente esta rotina retira a(s) tarefa(s) que gerou(raram) o problema da lista "osdead" recompondo sua estrutura.

  O núcleo não influencia o sistema no caso de estouro; entretanto o usuário será avisado através da chamada da rotina "osestouro(pid)", onde pid identifica a tarefa com problema.

  O usuário pode, por exemplo, optar por encerrar o processamento do sistema chamando a rotina TERMINATE PROGRAM.

  Durante a configuração o usuário será questionado da conveniência de utilizar "osestouro(pid)"; caso ele decline de sua utilização o núcleo seguirá com o processamento normal do sistema.

4.4.3. Escalonamento "Least Laxity First" - LAXMOSI

A descrição deste escalonamento seguirá a mesma sequência adotada na descrição do algoritmo "earliest deadline first", permitindo assim uma comparação entre as duas estratégias.

O núcleo gerado nesta implementação foi denominado LAXMOSI.

a) *As tarefas do núcleo.* Como no núcleo "DEADMOSI",as tarefas foram divididas em três classes: periódica com "deadline", não periódica com "deadline" e não periódica com prioridade.

   As tarefas não periódicas com "deadline" são aquelas que ocorrem no sistema em tempos indeterminados. Estas tarefas possuem como característica um valor inteiro positivo denominado "deadline", além de um valor inteiro positivo denominado tempo de computação.

   O "deadline" representa o tempo *D* no qual a execução da tarefa deve ter sido encerrada, sob pena de prejudicar o funcionamento normal do sistema.

   O tempo de computação representa o tempo que a tarefa deve utilizar o processador sendo fornecido em "ticks" do sistema.

b) *O algoritmo.* O algoritmo de escalonamento está implementado em dois níveis: com "deadline" e sem "deadline".

   No primeiro nível as tarefas são ordenadas em ordem crescente de LLS ("Least Laxity scheduling").

   O LLS é obtido pela diferença do "deadline" da tarefa e seu tempo de computação. Dessa forma, ele representa a "folga" que a tarefa possui antes que seu "deadline" seja atingido.

   Esse valor deve ser decrementado a cada "tick" para as tarefas que não estejam ocupando a CPU, pois para tais tarefas a "folga" está decrescendo.

Uma vez que não existam mais tarefas no primeiro nível o algoritmo continua a sua pesquisa no segundo nível, onde se encontram as tarefas ordenadas por prioridade.

Uma vez definidos os níveis é necessário alocar os tipos de tarefas definidas no item anterior ao nível apropriado.

As tarefas não periódicas com prioridade serão alocadas no nível dois, de acordo com a prioridade definida na sua criação.

As tarefas não periódicas com "deadline" e as tarefas periódicas com "deadline" serão naturalmente alocadas no nível um, de acordo com seus LLS.

c) *Características da implementação.* A estrutura modular do núcleo QUASIMOSI facilitou a implementação das alterações exigidas pelo novo algoritmo de escalonamento.

Nos próximos itens apresentaremos as principais mudanças ocorridas.

- *A lista "osdead".* O núcleo deve acompanhar o progresso das tarefas que possuem "deadline" de modo a poder determinar se os limites de tempo são obedecidos garantindo a integridade do sistema.

  Este acompanhamento é realizado a toda interrupção associada ao tick do sistema, onde as tarefas com "deadline" são verificadas para observar se houve ou não estouro do limite de execução.

  Assim, após cada interrupção de tempo a lista "osdead" é percorrida e o valor LLS que está armazenado no campo "pchave" do descritor de tarefas é decrementado representando uma diminuição na "folga" de tempo presente para a execução da tarefa.

  Quando o valor LLS chega a zero e a tarefa continua presente no sistema fica caracterizado a ocorrência de um "deadline".

  A variável "osdead" mantém o ponteiro para o ínicio desta lista.
  Para manter esta lista foram desenvolvidas duas rotinas de apoio,

inseridas no módulo proc.c: "osinsdead()" e "osretdead()".

É interessante salientar que o valor LLS é decrementado para todas as tarefas com "deadline" com exceção da tarefa que ocupa a CPU. Este fato inviabiliza a aplicação da técnica "lista delta", pois sua utilização implica em operações simultâneas sobre todas as tarefas da lista.

Naturalmente, a tarefa em execução poderia ser retirada da lista; entretanto, o custo computacional dessa manipulação não justifica sua implementação.

- *O descritor de tarefas (PID).* O descritor de tarefas é responsável pelo controle de toda atividade relacionada a uma tarefa.

Dentro deste descritor foram inseridos e/ou alterados os seguintes campos:

ptipo - indica o tipo ao qual a tarefa está associada, podendo assumir os seguintes valores: "deadline", periódica, e prioridade.

ppamtipo - indica o valor do parâmetro associado ao tipo. De acordo com o tipo este parâmetro assume os valores descritos na tabela XV..

Tabela XV. Tipos de tarefas.

TIPO	PARÂMETRO
DEADLINE	deadline associado
PERIÓDICA	deadline associodo
PRIORIDADE	prioridade associada

pdead - Aponta para o próximo elemento na lista "osdead", que mantém todas as tarefas com "deadline".

pchave - Mantém a chave de ordenação das tarefas do núcleo; no caso de tarefas com prioridade este campo contém a prioridade das tarefas; nos outros casos ele contém o LLS("least laxity scheduling"), que é obtido através da diferença entre o "deadline" e o tempo de computação da tarefa.

pinicio - Ponteiro para ínicio da tarefa.

pperiodo - contém o período no caso de tarefas periódicas.

pprxper- mantém o valor do relógio da próxima ativação de tarefas periódicas; este valor é utilizado na reinicialização da tarefa no período seguinte.

- *As listas do núcleo.* Nesta implementação do núcleo sempre que uma tarefa deve aguardar por um recurso ela é inserida numa lista associada ao recurso em questão.

  Esta inserção obedece à "importância" da tarefa no sistema; tal informação é fornecida pelo "deadline" ou pela prioridade.

  Entre os recursos gerenciados por listas temos: os semáforos, as "mailboxes", e o processador.

  De forma análoga às listas descritas para o algoritmo "earliest deadline first", nesta estratégia as listas passaram a possuir dois níveis: o primeiro das tarefas LLS, e o segundo das tarefas com prioridade.

  A inserção de tarefas com "deadline" obedece à sequência estabelecida através da entrada "pchave" presente no descritor de tarefas, ou seja, quanto menor for o valor de "pchave" mais próximo do ínicio da lista a tarefa se encontra.

  A inserção de tarefas com prioridade obedece a prioridade definida na sua criação.

- *A criação e destruição de tarefas.* Durante a criação de uma tarefa seu tipo é analisado e, caso ela pertença à classe de tarefas com "deadline", o ato de criação é acrescido de algumas ações que irão viabilizar sua implementação.

Uma das primeiras ações a serem executadas é a inserção da tarefa na lista "osdead" através da rotina "osinsdead()".

Outra ação está relacionado com o cálculo do valor da entrada "pchave" presente na estrutura descritora de tarefas, que é realizado através da seguinte operação: pchave = ppamtipo - tempo_comp.

No caso de destruição da tarefa, seu tipo é analisado e, caso ela pertença à classe de tarefas com "deadline", ela deve ser retirada da lista "osdead" através da rotina "osretdead()".

Além disso, caso a tarefa seja periódica, ela não é destruída no final de sua execução, mas sim "congelada" até o próximo periodo.

- *Estouro de "deadline".* No caso de estouro de "deadline" o sistema retira a(s) tarefa(s) que gerou(ram) o problema da lista "osdead" recompondo sua estrutura.

O núcleo não influencia o sistema no caso de estouro; entretanto, o usuário será avisado através da chamada da rotina "osestouro(pid)", onde pid identifica o tarefa com problema.

O usuário pode, por exemplo, optar por encerrar o processamento do sistema chamando a rotina TERMINATE PROGRAM.

Durante a configuração o usuário será questionado da conveniência de utilizar "osestouro()"; caso ele decline de sua utilização o núcleo seguirá com o processamento normal do sistema.

4.5. Exemplos Comparativos

Escalonadores preemptivos com prioridade estática são utilizados largamente em sistemas "Hard Real Time"; entretanto, existem ambientes onde o modelamento realizado com esses escalonadores não utiliza plenamente o processador.

A utilização de prioridades fixas torna o escalonador mais rígido e, portanto, mais lento na sua capacidade de se adaptar a possíveis mudanças no ambiente.

Como forma de ilustrar o problema citado vamos utilizar o exemplo fornecido por Howard Falk [FAL88b] aplicado a duas estratégias de escalonamento desenvolvidas neste trabalho: DEADMOSI x QUASIMOSI.

O exemplo ilustra um sistema que obtém e processa dados de dois sensores: *A* e *B*.

O período de coleta do sensor *A* é de 20*ms*, e o tempo utilizado no processamento dos dados é 10*ms*, e seu "deadline" é igual ao período, ou seja, 20*ms*.

O período de coleta do sensor *B* é de 50*ms*, seu período é de 50*ms*, e o tempo utilizado em seu processamento é de 25*ms*, e seu "deadline" é igual ao período, ou seja, 50*ms*.

O fator de utilização do processador (Definição 1 , capítulo 3) para o conjunto de tarefas acima é igual à:

$$U(2) = \sum_{i=1}^{2} (C_i/T_i) = 10/20 + 25/50 \Rightarrow U(2) = 1$$

Utilizando a expressão fornecida pelo Teorema 1 do capítulo 3 obtemos que, mesmo utilizando o algoritmo taxa monotônica não é possível cumprir o "deadline" das tarefas, já que para 2 tarefas o fator de utilização do processador deve ser menor ou igual a 0.83.

No entanto, se utilizarmos a estratégia "earliest deadline first" o fator de utilização do processador deve ser menor ou igual a 1, restrição que o conjunto de tarefas fornecido obedece.

O resultado obtido, com os algoritmos de escalonamento implementados, para o conjunto de tarefas acima pode ser observado na Figura 4.13..

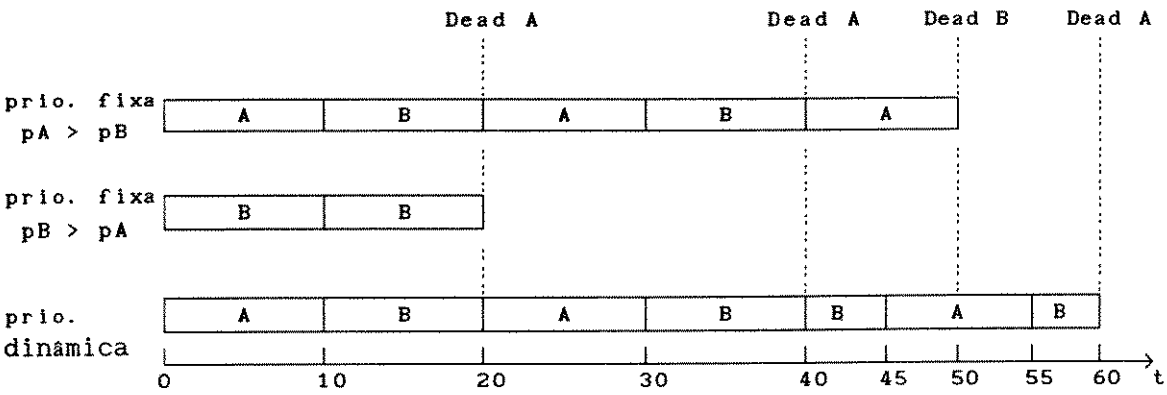

Figura 4.13. Comparação Fixa X Dinâmica

Foram implementadas duas estratégias com prioridade fixa e uma com prioridade dinâmica.

Na primeira estratégia, a técnica com prioridade fixa foi adotada com a prioridade da tarefa *A* maior que da tarefa *B*.

Nesse caso a tarefa *A* inicia a execução em t=0, ao seu final a tarefa *B* entra em execução em t=10 e executa até o próximo ciclo da tarefa *A*, que ocorre em t=20.

Nesse momento a tarefa *B* é suspensa e cede seu lugar para a tarefa *A* que executa até o seu final em t=30.

Ao final do segundo ciclo da tarefa *A* a tarefa *B* retoma sua execução permanecendo com o controle do processador até o terceiro ciclo da tarefa *A*, que ocorre em t=40, quando então cede o processador para a tarefa *A* que possui maior prioridade.

A tarefa *A* executa até o seu término que ocorre em t=50. Nesse momento ocorre o estouro do deadline da tarefa *B*, fato que encerra o processamento e invalida a estratégia.

Na segunda estratégia, permanece a técnica com prioridade fixa, sendo a prioridade da tarefa *B* maior que da tarefa *A*. Nesse caso a tarefa *B* ganha o processador e permanece executando até t=20, quando ocorre o estouro de "deadline" da tarefa *A* inviabilizando novamente a estratégia.

Na terceira estratégia a técnica de prioridade dinâmica é adotada utilizando-se a estratégia de escalonamento "Earliest Deadline First".

Nessa estratégia a "importância" da tarefa é determinada pelo seu "deadline", assim no tempo t=0 o "deadline" da tarefa *A* é dA=20 e o da tarefa *B* é dB=50, com isso, a tarefa *A* ganha o processador e executa até o seu final em t=10 quando então a tarefa *B* inicia a sua execução.

A tarefa *B* permanece em execução até t=20 quando é iniciado um novo ciclo de processamento para a tarefa *A*, sendo seu "deadline" igual a dA=40.

Como o "deadline" da tarefa *A* é mais crítico que da tarefa *B*,ela ganha o processador e executa até o seu final em t=30 quando então, a tarefa *B* continua sua execução.

No tempo t=40 inicia-se o terceiro ciclo da tarefa *A* sendo seu "deadline" dA=60, como o "deadline" da tarefa *B* é mais crítico do que o "deadline" dessa nova instância da tarefa *A*, a tarefa *B* continua sua execução até encerrar sua execução em t=45 dentro dos limites de seu "deadline".

Nesse exemplo pode-se observar que os algoritmos com prioridade dinâmica são mais versáteis do que o algoritmo de prioridade fixa, fato que garantiu que os "deadlines" das tarefas *A* e *B* fossem respeitados.

O programa utilizado neste teste é representado a seguir:

```
typedef struct {                    /* estrutura descr. da tarefa */
     funcid_t  tarefa;
     short     periodo;
     int       escalonador;
} pacote;

void tempo_execucao(int nro_tick)
{
     /* simula execucao da tarefa */
}

void gerente(void)
{
     pacote *descritor;

     osgtpin(&descritor, ...);          /* obtem param ativacao*/

     for (cont=0; cont<NRO_EXECUCAO; cont++){

         pid = oscrepr(descritor->tarefa, ....);   /* ativa tsk*/
         osdelay(descritor->periodo);     /* susp até prx ativ */
         estado = osgtpst(pid, ...);           /* obtém estado */
         if (estado != PRFREE) {       /* se tsk ativa => erro */
               printf("ESTOURO DE DEADLINE");

     }
}

void task_A(void);
{

     tempo_execucao(10);          /* simula execucao da tarefa */

}

void task_B(void);
{

     tempo_execucao(25);          /* simula execucao da tarefa */

}


void main(void)
{
     pacote descritor;

     pid_main = osintpg( ... );                /* ativa nucleo */

     descritor.tarefa = task_B;
     descritor.periodo = 50;
     descritor.escalonador = ESCALO_B;     /* depende tipo escalonamento */
     oscrepr(gerente, ... , &descritor , ...);
     descritor.tarefa = task_A;
     descritor.periodo = 20;
     descritor.escalonador = ESCALO_A;     /* depende tipo escalonamento */
     oscrepr(gerente, ... , &descritor , ...);
     ossuspr(pid_main);                /* encerra processamento */
}
```

Apesar das técnicas de escalonamento "Earliest Deadline First" (EDF) e "Least Laxity Scheduling" (LLS) serem ótimas, o resultado de seu escalonamento não é sempre o mesmo.

Para ilustrar este fato, vamos utilizar três tarefas com as características descritas na tabela XVI..

Tabela XVI. Descrição das tarefas.

TAREFA	DEADLINE	TEMPO COMPUTAÇÃO
T1	20	5
T2	25	5
T3	30	16

As figuras 4.14. e 4.15. atestam que as estratégias EDF e LLS não apresentam o mesmo padrão de comportamento. Note que na figura 4.14. a primeira tarefa a executar é T_3 pois a estratégia LLS garante a execução para a tarefa com a menor "folga" entre seu "deadline" e o tempo de execução; já na figura 4.15., com a estratégia EDF, a primeira tarefa a executar é a tarefa T_1 que possui o "deadline" mais crítico. O fator de utilização do processador para o conjunto de tarefas fornecidas na tabela XVI é igual a:

$$U(3) = \sum_{i=1}^{3} (C_i/T_i) = 5/20 + 5/25 + 16/30 \Rightarrow U(3) = 0.98333$$

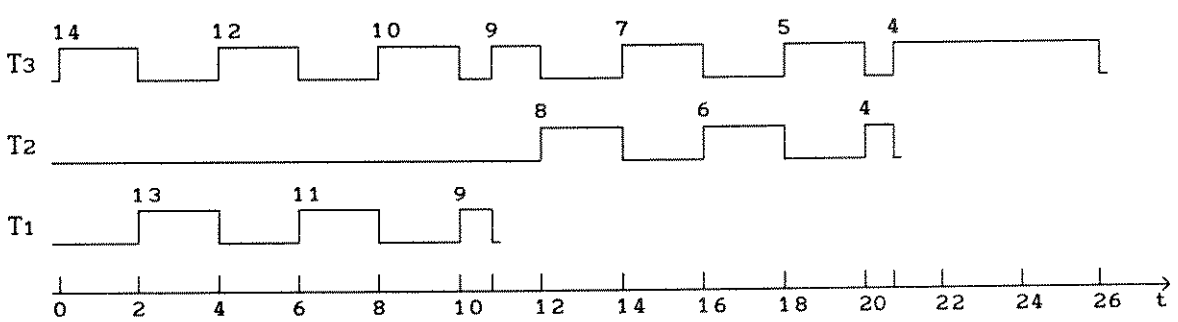

Figura 4.14. Algoritmo "Least Laxity Scheduling" (LLS).

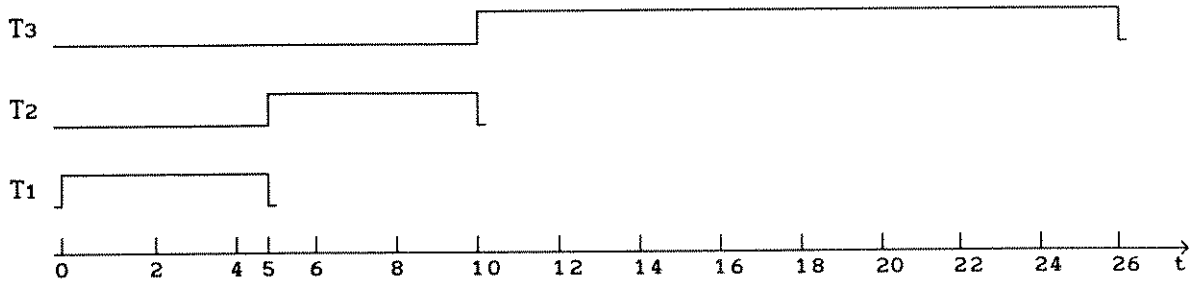

Figura 4.15. Algoritmo "Earliest Deadline First" (EDF).

Neste exemplo é possível observar uma desvantagem na utilização da estratégia LLS que consiste no grande número de comutações de contexto necessárias para garantir o "deadline" de cada tarefa.

Outro ponto negativo reside na maior complexidade de implementação da estratégia LLS que exige a cada "tick" do sistema, operações de subtração sobre as tarefas que não ocupam a unidade de processamento.

Uma das mais severas restrições na utilização das estratégias EDF e LLS consiste na exigência de que os processos sejam independentes para garantir que elas sejam ótimas [MOK84].

Esse fato é facilmente observado no exemplo descrito pela tabela XVII..

Tabela XVII. Descrição das tarefas.

TAREFA	DEADLINE	TEMPO COMPUTAÇÃO
T1	20	10
T2	25	5
T3	30	8

Sendo que a tarefa T_1 no tempo t=5 aguarda uma mensagem enviada pela tarefa T_3 ao final de sua execução. Note que neste exemplo não podemos aplicar o teorema 1 ou o teorema 2 pelo fato das tarefas não serem independentes.

Com o escalonamento EDF obteremos a resposta apresentada na figura 4.16, onde podemos observar que devido a uma sincronização entre as tarefas, realizada através da troca de mensagens, a tarefa T_1 não consegue respeitar

suas restrições de tempo gerando um estouro de "deadline" em t=20.

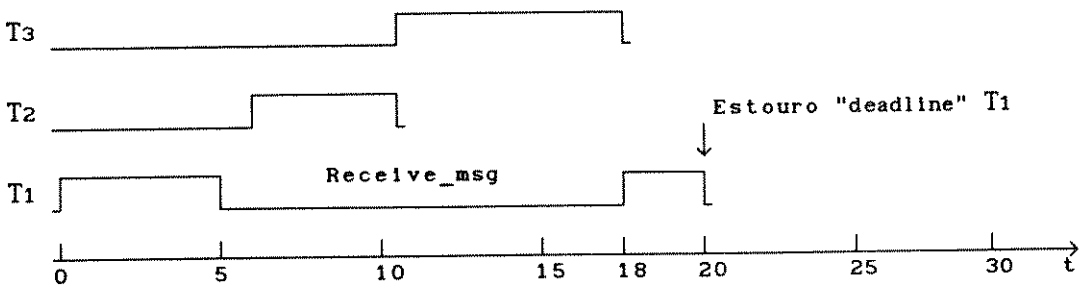

Figura 4.16. Estouro "deadline" T_1.

Um dos motivos do estouro de deadline da tarefa T_1 é a sua suspensão permanente aguardando a mensagem de T_3. Observa-se que se a função RECEIVE_MSG fosse temporizada (por exemplo, por 0.5 "ticks") o sistema seria viável, pois essa temporização pode ser adicionada ao tempo de computação da tarefa T_1 e o conjunto de tarefas ser considerado independente.

Dessa forma uma alternativa simples para aumentar a classe de problemas nos quais os algoritmos LLS e EDF são ótimos é exigir que toda interação entre as tarefas seja temporizada, ou em outras palavras, garantir a independência entre tarefas.

5. CONCLUSÃO E PERSPECTIVAS

Neste trabalho foram abordados os diversos elementos que devem ser considerados em sistemas de tempo real com severas restrições de tempo, com atenção especial para escalonadores de tarefas.

Como forma de sedimentar os conhecimentos adquiridos, além de atender as necessidades existentes na área, foi gerado um núcleo de tempo real cujas características mais importantes serão novamente ressaltadas:

- Facilidade para implementação de algoritmos de escalonamento, sendo geradas versões para:
  - prioridade fixa,
  - "earliest deadline first",
  - "least laxity first".

- Aderente à norma MOSI (Microprocessor Operating Systems Interfaces) da IEEE.

- Implementado em linguagem de programação de alto nível.

O produto resultante desta implementação está sendo utilizado em projetos internos dentro do CTI-IA, como por exemplo, na implementação de um sistema de comunicação industrial [ALM89] baseado no protocolo Mini-MAP (Manufacturing Automation Protocol) [GEN88].

Além de sua utilização dentro do CTI-IA, os núcleos gerados estão sendo utilizados como suporte para pesquisas na área em cursos de graduação e pós-graduação na UNICAMP-FEE.

Do ponto de vista da implementação, cinco pontos merecem ser destacados pelo fato de não serem observados em implementações tradicionais.

O primeiro está relacionado com a utilização prática de algoritmos de escalonamento para tempo real em implementações de núcleos. Apesar de tais algoritmos já existirem há alguns anos, somente agora eles passaram a fazer parte do horizonte de informações do usuário, fato que pode ser observado na sua recente utilização na construção de um sistema baseado na linguagem ADA [SHA90a].

O segundo está relacionado com a implementação do protocolo Limite de Prioridade [SHA90b] que, associado ao algoritmo Taxa Monotônica representa uma solução factível na implementação de muitos sistemas "Hard Real Time".

O terceiro trata da divisão das tarefas em duas classes, nas versões que levam em consideração o "deadline", de forma a fornecer um tratamento especial para tarefas que não possuam um "deadline" associado.

O quarto ponto destaca a estratégia adotada na organização das listas que mantêm as tarefas que estão aguardando recursos, onde a inserção de uma tarefa leva em consideração a importância que a mesma possui dentro do sistema.

Finalmente, temos o configurador do núcleo que, através de um diálogo com o usuário, gera exatamente uma biblioteca que atende às necessidades da aplicação.

Os algoritmos implementados neste trabalho compõem uma base importante para o estudo de sistemas HRT; no entanto, eles representam apenas o ínicio de um trabalho muito mais amplo de aquisição de tecnologia na área.

No futuro, muitos dos sistemas de tempo real serão grandes, complexos e implementados em sistemas distribuídos [STA88]. A gerência desses sistemas exigirá um grande esforço de desenvolvimento em áreas que compreendem desde a especificação do sistema até as técnicas que viabilizem a implementação, tanto do ponto de vista do software quanto de hardware.

Como forma de concatenar todos esses elementos, apresentaremos um modelo para gerar algoritmos de escalonamento derivado do modelo desenvolvido por Ma et al [MA82].

Basicamente, o objetivo é analisar o sistema, desde a fase de análise, para determinar características que determinem qual algoritmo é mais apropriado para a aplicação específica.

A figura 5.1 apresenta a estrutura proposta.

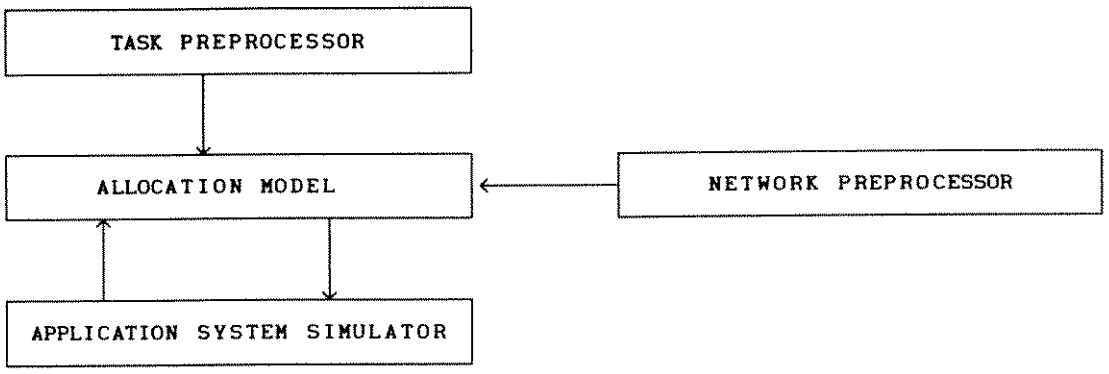

Figura 5.1. Estrutura Geral do Analisador de STR.

O "task preprocessor" é responsável em determinar características das tarefas que fazem parte do sistema tais como:

- relação de precedência
- fator de acoplamento
- tamanho da tarefa
- fatores de temporização, etc.

O "Network Preprocessor" analisa o hardware onde o sistema deverá ser mantido fornecendo parâmetros para gerar o modelo de alocação, tais como:

- distância entre os processadores
- custo de comunicação
- velocidade de processamento
- quantidade de recursos disponíveis, etc.

O "Allocation Model" determina qual modelo é mais apropriado para as características do sistema, procurando atender a todos os requisitos do sistema, sempre dando preferência a algoritmos ótimos, para posteriormente verificar os sub_ótimos.

Finalmente o "Application System Simulator", tem por objetivo validar o algoritmo obtido no "Allocation Model" , possibilitando uma realimentação, quando as necessidades do usuário não são completamente atendidas.

Esse modelo comprova a complexidade existente no tratamento de sistemas de tempo real com severas restrições de tempo; assim, algoritmos de escalonamento representam apenas um tópico no esforço de desenvolvimento das bases científicas desses sistemas; entretanto, o trabalho nessa área não é pequeno.

O objetivo é obter algoritmos que possibilitem o tratamento de complexas estruturas de tarefas com precedência e recursos limitados associadas a restrições de tempo.

7. BIBLIOGRAFIA

[ALM89] Almeida H. J., Appezato L., Araujo O. F. N., Faria M. M., Koyama M. F., Pereira E. D., Ribeiro I. M. C., Tavares R. F., "Implementação de um Sistema de Comunicação Industrial no CTI", Seminário Franco-Brasileiro em Sistemas Informáticos Distribuídos, Florianópolis, SC., 11-14 set, 1989.

[AZE86] Azevedo H., "NTR86 - Núcleo de Tempo Real", Instituto de Cooperacion Iberoamericana, Programa de Ciencia y Tecnologia para el Desarollo , Informe III semestre, Mexico - DF, junho, 1986.

[AZE89a] Azevedo H., Executivo de Tempo Real DEADMOSI - Manual do Usuário, DTIA 00289, Instituto de Automação, Centro Tecnológico para a Informática, out 1989.

[AZE89b] Azevedo H., Executivo de Tempo Real QUASIMOSI - Manual do Usuário, DTIA 00189, Instituto de Automação, Centro Tecnológico para a Informática, jun 1989.

[AZE90a] Azevedo H., Magalhães M.,"Técnicas de Escalonamento para Sistemas "Hard Real Time"", IV Congresso Nacional de Automação Industrial - CONAI, São Paulo, SP., 23-27 jul, 1990.

[AZE90b] Azevedo H., Silva Jr. A. F., Magalhães M., "Implementação de um Núcleo "Hard Real Time" aderente a Norma MOSI da IEEE",I Simpósio de Automação Integrada do CEFET-PR, Curitiba, PR., 17-21 jul, 1990.

[CHE86] Cheng S. C., Stankovic J. A., Ramamritham K., "Dynamic Scheduling of Groups of Tasks with Precedence Constraints in Distributed Hard Real_Time Systems", IEEE Real Time System Symposium, 1986.

[CHE87] Cheng S. C., Stankovic J. A., Ramamritham K., "Scheduling Algorithms for Hard Real Time System - A Brief Survey", Real_Time System Newsletter, 2(3), summer 1987.

[COE86] Coelho J. M. A., Suporte de Tempo Real para um Ambiente de Programação Concorrente, Dissertação de Mestrado em Engenharia Elétrica, FEE UNICAMP, agosto, 1986.

[COM85] Comer D., Operating System Design The XINU Approach, Englewood Cliffs - New Jersey, Prentice-Hall, 1985.

[COR62] Corbato F.J., Merwin-Dagget M., Daley, R. C., "An Experimental Time-Sharing System", Proc AFIPS Fall Joint Comput Conj, 335-344, May 1962.

[EFE82] Efe K., "Heuristic Models of Task Assignment Scheduling in Distributed System", Computer, jun, 1982.

[ELS82] Elsayed E. A., "Algorithms for Project Scheduling with Resource Constraints", International Journal of Production Research, 1982.

[FAL88a] Falk H.,"CASE Tools emerge to handle Real-Time Systems", Computer Design, jan, 1988.

[FAL88b] Falk H., "Developer Target Unix and Ada with Real Time Kernels, Computer Design, april, 1988.

[FRE86] French S., Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop, Ellis Horwood Limited, 1986.

[GAR75a] Garey M. R., Graham R. L., "Bounds for Multiprocessor Scheduling with Resource Constraints", SIAM J. Comput., 2(4), jun, 1975.

[GAR75b] Garey M. R., Johnson S., "Complexity Results for Multiprocessador Scheduling Under Resource Constraints", SIAM J. Comput., 4(4), dec, 1975.

[GAR79] Garey M. R., Johnson D. S., Computers and Intractability, A Guide to Theory of NP-Completeness, San Francisco, C.A., Freeman, 1979.

[GEN88] General Motors Co., Manufacturing Automation Protocol (MAP) Specification, versão 3.0, aug, 1988.

[HAN78] Hansen P. B., "Distributed Process: a Concurrent Programming Concept", CACM, 11(21):934-941, nov, 1978.

[HAN73] Hansen P. B., Operating System Principles, Prentice- Hall, 1973.

[HEX88] Hexel R.A., Núcleo Multiprocessado para Aplicações em Tempo Real, Dissertação de Mestrado em Ciência da Computação, IMECC UNICAMP, agosto, 1988.

[HOA74] Hoare C. A.R.,"Monitors: An Operating System Structuring Concept, CACM, 10(17):549-557, out, 1974.

[HOR74] Horn W. A.,Simple Scheduling Algorithms, Naval Research Logistics Quarterly, 1974.

[HOW88] Howard, F., "CASE Tools Emerge to Handle Real-Time Systems", Computer Design, jan, 1988.

[HWA84] Hwang, K., Fayé, A. B., Computer Architecture and Parallel Processing, McGraw-Hill, 1984

[IEE87] IEEE Project 855., Draft Standard for Microprocessor Operating System Interfaces, The IEEE Inc., 1987, Draft 7.

[KLE70] Kleinrock, L.,"A Continuum of Time Sharing Scheduling Algorithms" Proc AFIPS 36, SJCC ,452-458, 1970.

[KNU73] Knuth D. E., The Art of Computer Programming, vol 1, Addison-Wesley, Massachusetts, 1973.

[LAI84] Lai T. H., Sahni S., "Preemptive Scheduling of a Multiprocessor System with Memories to Minimize Maximum Lateness", SIAM J. Comput., 4(13), nov, 1984.

[LAW73] Lawler E. L., "Optimal Scheduling of a Single Machine Subject to Precedence Constraints", Management Science 1973.

[LIU73] Liu C. L., Layland J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment", Journal of the Association for Computing Machinery, 1(20):46-61, jan, 1973.

[LOC75] Locke C. D., Tokuda H., Jensen E. D., A Time-Driven Scheduling Model for Real-Time Operating Systems, Technical Report, Carnegie_Mellon University, 1975.

[MAD74] Madnik S.E., Donavan J.J., Operating System, McGraw- Hill, 1974.

[MA82] Ma R. P.-Y., Lee E. Y. S., Tsuchiya M., "A Task Allocation Model for Distributed Computing Systems", IEEE Transactions on Computers, 1(C31), jan, 1982.

[MA84] Ma R. P.Y.,"A Model to Solve Timing_Critical Application Problems in Distributed Computer Systems", Computer, jan, 1984.

[MAG90] Magalhães M. F., Sistemas de Tempo Real, Depto. de Eng. da Comp. Automação Industrial, FEE - UNICAMP, 1990.

[MOK78] Mok A. K., Dertouzos H. L., "Multiprocessor Scheduling in a Hard Real-Time Environment", Proc. of Seventh Texas Conference on Computing Systems, nov, 1978.

[MOK84] Mok A. K., "The Design of Real-Time Programming based on Process Models", IEEE Software, 1984.

[MOO86] Mooney, J.D.,"Lessons from the MOSI Project",Proceedings Computer Standard Conference 1986, San Francisco, CA., 54-61, may 13-15, 1986.

[MOO69] Moore J. M., "An n Job, One Machine Sequencing Algorithm for Minimize the Number of Late Jobs", Management Science, 1969.

[MUN70] Muntz R.R., Coffman Jr. E.G., "Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems", Journal of the Association for Computing Machinery, 2(17), april, 1970.

[RAM84] Ramamritham K., Stankovic J. A., "Dinamic Task Scheduling in Distributed Hard Real-Time System", IEEE Software, vol 1, jul, 1984.

[RAY87] Rayward-Smith V.J.,"The Complexity of Preemptive Scheduling given Interprocessor Communication Delays", Information Processing Letters, 2(25), may, 1987.

[SAD87] Sadayappan P.,Ercal F.,"Nearest-Neighbor Mapping of Finit Element Graphs onto Processor Meshes", IEEE Transactions on Computers, 12(C36), dec, 1987.

[SHA90a] Sha L., Goodenough J. B., "Real-Time Scheduling Theory and Ada", Computer, 53-62, april, 1990.

[SHA90b] Sha L., Rajkumar R., Lehoczky J. P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", IEEE Transactions on Computers, 9(39):1175-1185, setembro, 1990.

[SPR89] Sprunt B., Sha L., Lehoczky J., "Aperiodic Task Scheduling for Hard-Real-Time Systems", J. Real-Time Systems, 1(1):27-60, 1889.

[STA88] Stankovic J. A., "Misconceptions about Real-Time Computing - A Serious Problem for Next Generation Systems", IEEE - Computer, october, 1988.

[SUY75] Suydan W. E. Jr.,"Off-the-shelf Software tackles Real Time Task", Computer Design, 65-70, july, 1975.

[TAN87] Tanenbaum, A.S., Operating System: Design and Implementation, Prentice-Hall, 1987.

[TSI74] Tsichritzis D.C., Bernstein, Operating System, Academic Press, 1974.

[WAR85] Ward P., Mellor S., Structured Development for Real_Time Systems, New Jersey, Prentice-Hall, 1985.

[ZHA87a] Zhao W., Ramamritham K., Stankovic J. A., "Preemptive Scheduling Under Time and Resource Constraints", IEEE Transactions on Computers, 8(C36), aug, 1987.

[ZHA87b] Zhao W., Ramamritham K., Stankovic J. A., "Scheduling Tasks with Resource Requerements in Hard Real-Time System", IEEE Transactions on Software Engineering, 5(SE13), may, 1987.

ANEXO A

TEMPOS E TAMANHOS DOS NÚCLEOS

A.1. TEMPOS E TAMANHOS DOS NÚCLEOS

A1.1. Tempos

Na tabela A.I. são apresentadas as medidas relacionadas com o tempo de execução de algumas funções dos núcleos implementados.

Os tempos foram obtidos com analisador lógico "Tektronix DAS 9100" utilizando o módulo 91A24 com clock de 100ns, sendo o micro utilizado na medição, o SP16 da Prológica ativado a 4Mhz na sua configuração padrão.

Para maior clareza, serão fornecidos a seguir trechos dos programas utilizados na aquisição dos tempos. A sintaxe utilizada na aquisição dos tempos segue a linha do núcleo QUASIMOSI; entretanto, a sequência é a mesma para todos os núcleos.

a) Criar uma tarefa

```
     |                                          /* prioridade 1 */
proc1  ( )                                      /* término contagem */
{
...                                             /* prioridade 3 */
}
main   ( )
{
pid1=oscrepr(proc1,true,1,800,NULL,0,&erro);/* início contagem */
     |
}
```

b) Comutação de tarefa

```
     |                                          /* prioridade 3 */
proc1 ( )                                       /* término contagem */
{
...                                             /* prioridade 1 */
}
main   ( )
{
pid_main = osintpg (1,0x300,&erro);
pid1 = oscrepr (proc1, FALSE ,3,800,NULL,0,&erro);
erro = osrespr (pid1);                          /* início contagem */
erro = ossuspr (pid_main);
     |
}
```

c) Criar um semáforo

```
main  ( )
{
pid_main = osintpg (3,0x300,&erro);          /* início contagem */
sem1= osallsm (10,&erro);                    /* término contagem */
if (erro != ok)
printf ("erro = % d", erro);
     |
 }
```

d) Sinalizar um semáforo

```
main  ( )
{
pid_main = osintpg (3,0x300,&erro);
sem1 = osallsm (10, & erro);                 /* início contagem */
erro = ossigsm (sem1);                       /* término contagem */
if (erro != ok);
    printf ("erro = % d\n", erro)
     |
 }
```

e) Alocar bloco de memória

```
main  ( )
{                                            /* início contagem */
     |                                       /* término contagem */
block = osalmem (500);
if (block == NULL)
   printf ("ERRO DE MEMÓRIA") ;
     |
 }
```

Tabela A.I. Tempos de execução das funções.

TABELA DOS TEMPOS DE EXECUÇÃO DOS NÚCLEOS			
AÇÃO	QUASIMOSI	DEADMOSI	LAXMOSI
Criar tarefa c/ prioridade	2,76ms	2,98ms	2,98ms
Criar tarefa c/ "deadline"	-	3,09ms	2,86ms
Criar tarefa periódica	-	3,09ms	2,86ms
Comutação de processos	706us	727us	728us
Criar um semáforo	374us	372us	372us
Sinalizar um semáforo	269us	272us	272us
Alocar bloco de memória	578us	576us	576us

A.2. Tamanhos

Nas tabelas A.II., A.III. e A.IV são apresentados os tamanhos dos núcleos QUASIMOSI, DEADMOSI E LAXMOSI respectivamente.

Todos os elementos fornecidos foram obtidos na implementação com IBM/PC, com os valores de constantes do arquivo config.h iguais ao valor default.

Tabela A.II. Área do núcleo QUASIMOSI.

ÁREA OCUPADA PELO NÚCLEO QUASIMOSI		
MÓDULO	TAMANHO	
CONFIG.C PROC.C MEM.C SEM.C QUEUE.C MAIL.C TIMER.C HANDLER.C POOL.C EVENTO.C INTERF.S	394 982 65E 4EC 18E BC0 5F2 6FA 598 5F8 4BC	
TOTAL CÓDIGO	4232H	16,54K
TOTAL DADOS	A34H (FAR_BSS)	2,55K

Tabela A.III. Área do núcleo DEADMOSI.

ÁREA OCUPADA PELO NÚCLEO DEADMOSI		
MÓDULO	TAMANHO	
DCONFIG.C DPROC.C DMEM.C DSEM.C DQUEUE.C DMAIL.C DTIMER.C DHANDLER.C DPOOL.C DEVENTO.C DINTERF.S	3C4 10C6 676 500 230 B04 5F2 874 598 5F8 57A	
TOTAL CÓDIGO	4CA4H	19,16K
TOTAL DADOS	B78H (FAR_BSS)	2,86K

Tabela A.IV. Área do núcleo LAXMOSI.

ÁREA OCUPADA PELO NÚCLEO LAXMOSI		
MÓDULO	TAMANHO	
DCONFIG.C DPROC.C DMEM.C DSEM.C DQUEUE.C DMAIL.C DTIMER.C DHANDLER.C DPOOL.C DEVENTO.C DINTERF.S	398 E9C 616 4E8 22E A8E 5E8 82A 564 5F0 56D	
TOTAL CÓDIGO	48C1H	18,18K
TOTAL DADOS	BA7H (FAR_BSS)	2,91K

ANEXO B

TELAS DO CONFIGURADOR

CENTRO TECNOLÓGICO PARA A INFORMÁTICA CTI
INSTITUTO DE AUTOMAÇÃO
DIVISÃO DE PROCESSOS
HELIO AZEVEDO
JANEIRO/1990

CONFIGURADOR DO NÚCLEO DEADMOSI ver 1.1

Pressione qualquer tecla para continuar ...

```
-  O objetivo deste configurador é criar uma biblioteca do núcleo de tempo
-  real XXXXMOSI compatível com a sua aplicação.
-  O núcleo foi escrito em linguagem C e compilado pelo Compilador C da
-  Microsoft versão 5.0 no modelo LARGE.
-
-  Para que este programa atue corretamente é necessário que os seguintes
-  elementos estejam disponíveis:
-          1. Compilador C da Microsoft versão 5.0 ou compatível
-          2. Microsoft Macro Assembler versão 5.0 ou compatível
-          3. Sistema operacional MS-DOS versão 3.0 ou compatível
-          4. Variáveis de environment: PATH, INCLUDE, LIB definidas.
-          5. Núcleo DEADMOSI instalado com INSTALL
-          6. Diretório corrente seja ....\ATIVA
-
-
-  A operação total deverá durar no máximo 10 minutos.
-
-
-  Você deseja continuar ? (Y/n)


-  A biblioteca gerada será inserida no diretório: %DBIB% ,
-  e terá o nome de DMOSI.LIB
-
-  Esta localização é aceitável ? (Y/n)
```

CONFIGURADOR DE PROCESSOS

Este módulo é responsável pelo tratamento da "Capability Process Management" do núcleo DEADMOSI.
Através dele você poderá criar, destruir, e obter informações de processos presentes no sistema.
Naturalmente este módulo está sempre presente no núcleo, entretanto você pode alterar o número MÁXIMO de processos presentes no sistema em um dado instante.

Sua configuração apresenta os seguintes valores:
módulo presente
número máximo de processos = 20

Selecione uma das opções abaixo:
1. Configuração aceitável
2. Altera número de processos
3. Finaliza configurador

Selecione:

```
-
-                SALVAMENTO DOS REGS DO COPROCESSADOR
-
-
-  Associado a gerência de processos, está o salvamento de regis-
-  tradores do sistema durante a comutação de tarefas.
-  Este salvamento pode ser divido em duas classes: registradores
-  do microprocessador e registradores do coprocessador.
-  O salvamento dos registradores do micro é naturalmente realiza-
-  do a cada comutação de contexto, entretanto o salvamento dos
-  registradores do coprocessador é opcional.
-  Se você não utiliza operações de ponto flutuante, ou as isolou
-  em uma única tarefa, este salvamento não precisa ser realizado,
-  otimizando a comutação de tarefas.
-
-
-  Voce deseja salvar os regs do coprocessador ? (Y/n)
```

```
-
-
-                CONFIGURADOR DAS TEMPORIZAÇÕES DO SISTEMA
-
-
-  Este módulo é responsável pela manipulação do fator tempo no
-  núcleo DEADMOSI.
-  Através dele você poderá colocar um processo para dormir por um
-  determinado tempo, ou ainda, manipular timers existentes no sis-
-  tema.
-
-  Sua configuração apresenta os seguintes valores:
-            Módulo presente
-            número de timers extras = 1
-
-  Selecione uma das opções abaixo:
-            1. Configuração aceitável
-            2. Módulo ausente
-            3. Altera número de timers
-            4. Finaliza configurador
-
-  Selecione:
```

-
- Você optou pela utilização do módulo de temporizações do sistema.
- Resta ainda definir o período de ticks com o qual seu sistema deve trabalhar.
- O valor default do período de ticks é 54.9ms, que corresponde a 18,2 interrupções por segundo (padrão MS_DOS).
- Por motivos de programação do 8253 e compatibilidade com o MS_DOS somente deve-se utilizar valores que sejam divisores de 65536.
- Assim podemos criar a seguinte tabela:
-

FATOR_DE_DIVISAO	PERIODO	NUMERO_DE_INT_POR_SEG
1	54,900ms	18,21
2	27,450ms	36,42
4	13,725ms	72,85
8	06,862ms	145,73
16	03,431ms	291,46
32	01,715ms	582,92
64	00,858ms	1165,84

-
- Sua configuração apresenta o seguinte valor:
- fator de divisão = 1
-
- Pressione qualquer tecla para continuar ...

```
-
-
-  Sua configuração apresenta o seguinte valor:
-           fator de divisão = 1
-
-  Selecione uma das opções abaixo:
-           1. Fator aceitável
-           2. Altere fator
-           3. Finaliza configurador
-
-  Selecione:
```

-
- Em alguns sistemas de controle é necessário associar ao tick
- do sistema uma rotina para manipulação de dados.
- É evidente que poderiam ser utilizadas tarefas periódicas para
- resolver tais situações, entretanto, quando o período do tick é
- muito pequeno e a capacidade de processamento da máquina é res-
- trita essa solução pode ser inviável.
- Para resolver tais situações foi implementado uma forma menos
- elegante do que tarefas periódicas onde, na própria rotina de
- tratamento de interrupções de tempo é chamada uma função para
- realizar esse processamento.
- O endereço dessa rotina é definido pelo usuário através da fun-
- ção osfunctr().
- Essa função não pode realizar nenhuma chamada ao núcleo e deve
- ser o mais concisa possível.
-
-
-
-
-
-
-
- Pressione qualquer tecla para continuar ...

```
-
-                    CONFIGURADOR DE SEMÁFOROS
-
-
-  Este módulo é responsável pelo tratamento das "Capabilities
-  Modules Semaphores" e "Multiple Semaphores" integrantes do núcleo
-  DEADMOSI.
-  Através dele você poderá implementar sincronização entre proces-
-  sos através de operações elementares de sincronização do tipo:
-  signal e wait.
-
-  Sua configuração apresenta os seguintes valores:
-            módulo presente
-            número máximo de semáforos = 20
-
-  Selecione uma das opções abaixo:
-            1. Configuração aceitável
-            2. Módulo ausente
-            3. Altera número de semáforos
-            4. Finaliza configurador
-
-  Selecione:
```

```
-
-
-                    CONFIGURADOR DE MAILBOX
-
-
-
-  Este módulo é responsável pelo tratamento das "Capabilities
   Modules Messages" e "Messages with Responses" integrantes do
   núcleo DEADMOSI.
-  Através dele você poderá implementar troca de mensagens entre
-  processos através de operações de comunicação do tipo: send e
-  receive.
-
-  Sua configuração apresenta os seguintes valores:
-           módulo presente
-           número máximo de mailboxes = 20
-
-  Selecione uma das opções abaixo:
-           1. Configuração aceitável
-           2. Módulo ausente
-           3. Altera número de mailbox
-           4. Finaliza configurador
-
-  Selecione:
```

```
-
-                    CONFIGURADOR DE POOL DE MEMÓRIA
-
-
-   A "Capability Module Multispace Memory Management" presente na
-   "Capability Memory Management" é responsavel pelo tratamento de
-   Pools de memória dentro do núcleo DEADMOSI.
-   Através dela você pode obter e liberar blocos de memória de  uma
-   área especificada na definição do Pool, dessa  forma é possível
-   restringir a área onde o processo obtém blocos livres.
-
-   Sua configuração apresenta os seguintes valores:
-            módulo presente
-            número máximo de pools = 20
-
-   Selecione uma das opções abaixo:
-            1. Configuração aceitável
-            2. Módulo ausente
-            3. Altera número de pools
-            4. Finaliza configurador
-
-   Selecione:
```

```
-
-
-                    CONFIGURADOR DE EVENTOS
-
-
-  Este módulo introduz uma extensão a norma MOSI, para atender
-  a Interface de Aplicação do Protocolo MAP 3.0 .
-  Basicamente este modulo implementa semáforos clássicos com a
-  possibilidade de um processo aguardar a ocorrência de um conjun-
-  to de eventos.
-
-  Sua configuração apresenta os seguintes valores:
-            capability presente
-            número máximo de eventos = 20
-            número máximo de eventos na expressão de eventos = 8
-  A expressão de eventos mantém todos eventos aos quais o proces-
-  so está associado.
-
-  Selecione uma das opções abaixo:
-            1. Configuração aceitável
-            2. Módulo ausente
-            3. Altera número de eventos e/ou expressão de eventos
-            4. Finaliza configurador
-
-  Selecione:
```

```
-
-
-                    CONFIGURADOR DE EXCEÇÕES
-
-  A "Capability Exception Handling" define funções de habilitar o
-  o tratamento de exceções.
-  Exceções são eventos que exigem um tratamento que interrompe o
-  ciclo normal de processamento.
-  Para definir uma exceção é necessário que você indique quais ve-
-  tores do 80xxx serão utilizados e associá-los a classes e sub-
-  classes presentes na norma MOSI.
-  É altamente recomendável que o usuário não altere os vetores
-  de interrupção utilizados pelo BIOS ou pelo DOS (faixa ate 3fh).
-  O único vetor alterado pelo DEADMOSI é o vetor associado ao
-  'tick' do sistema (vetor 8h), logo se você desejar utilizar este
-  vetor deve manter a chamada do núcleo.
-
-  Sua configuração apresenta os seguintes valores:
-            módulo ausente
-
-  Selecione uma das opções abaixo:
-            1. Configuração aceitável
-            2. Módulo ausente
-            3. Altera configuração
-            4. Finaliza configurador
-
-  Selecione:
```

```
-
-  Você será questionado a respeito do vetores do 80xx que serão
-  utilizados, bem como as classes e subclasses aos quais serão
-  associados.
-  Não e' necessário que o usuário inicialize os vetores do micro,
-  pois essa operação será realizada pelo núcleo, entretanto o
-  usuário deve garantir que o dispositivo periférico que gerará
-  a interrupção seja programado de forma adequada.
-  Classes são simplesmente agrupamento de interrupções que serão
-  tratadas pelo mesmo "interrupt handler".
-  Subclasses servem para identificar uma interrupção que faz par-
-  te de uma classe dentro de um "interrupt handler".
-
-
-
-  Entre com o vetor:
-
-  Entre com a classe associada ao vetor x :
-
-  Entre com a subclasse associada ao vetor x e a classe y
-  Caso você não deseje definir subclasse entre: 00
-  Subclasse:
-
-
-  A seguinte configuração foi definida:
-
-           vetor = x   classe = y   subclasse = z
-
-  Selecione uma opção:
-           1. Aceita entrada e finaliza configuração de vetores
-           2. Repita entrada anterior
-           3. Define outra entrada
-           4. Finaliza Configurador
-
-  Selecione:
```

-
-
- Voce acabou de definir os vetores que serão utilizados pelo seu
- Sistema, resta somente definir o número de exceções que estarão
- ativas ao mesmo tempo no Sistema.
- Você pode utilizar o mesmo número de vetores que foram defindos
- ou então um valor menor para economizar memória. OK!
-

- Entre com o número de exceções:

```
-
-
-                    INICIALIZAÇÃO DO HARDWARE
-
-  Existem situações onde é necessário que certos elementos pre-
-  sentes no hardware tenham uma inicialização explícita pelo
-  usuário.
-  Para esses casos o núcleo DEADMOSI oferece a opção da chamada
-  de uma rotina (ini_user()) definida pelo usuário.
-  Essa chamada é realizada na inicialização do núcleo numa área
-  onde interrupções estão desabilitadas.
-
-  Selecione uma opcão:
-           1. Criar e utilizar ini_user()
-           2. Ignorar ini_user()
-           3. Finaliza Configurador
-
-  Selecione:
-
```

UTILIZACÃO DO TRATAMENTO DE ESTOURO DE DEADLINE

Quando ocorre o estouro de um deadline, o núcleo é informado e deverá, basicamente, retirar o processo da lista de deadlines recompondo sua estrutura.
Dessa forma o núcleo não influencia o sistema em caso de estouro, entretando o usuário pode ser avisado através da chamada da rotina criada pelo usuário denominada osestouro(pid), onde pid identifica o processo causador do problema.

Selecione uma opção:

1. Criar e utilizar osestouro()
2. Ignorar osestouro()
3. Finaliza Configurador

Selecione:

```
-
-
-
-  COMPILANDO OS ARQUIVOS DO NÚCLEO DEADMOSI....
-
-
-
-  Foi criada a biblioteca DMOSI.LIB e inserida no diretório %DBIB%.
-
-  Este núcleo é um protótipo e portanto passível de erros, no en-
-  tanto antes de 'acusar' o núcleo verifique se você está anali-
-  sando os códigos de erros retornados pelas rotinas do núcleo,
-  eles são a única interface do núcleo com você: NAO PODEM SER
-  IGNORADOS.
-
-  Não esqueça de colocar a opção /Gs quando compilando seus ar-
-  quivos escritos em linguagem C, esta opção inibe o check de pi-
-  lha pelo compilador, pois este check é realizado pelo DEADMOSI.
-
-                              BOA SORTE
```