UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

# " SSE: PROPOSTA DE UMA FERRAMENTA DE SOFTWARE DEDICADA AO ESTUDO DE ALGORITMOS DE ESCALONAMENTO PARA SISTEMAS DE TEMPO REAL CRÍTICO "

Autor: RODRIGO MARTINEZ SPANÓ

Orientador: Prof. Dr. MAURÍCIO FERREIRA MAGALHÃES

Dissertação apresentada à Faculdade de Engenharia Elétrica da Universidade de Campinas - FEE - UNICAMP, como parte dos requisitos exigidos para a obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA.

DEZEMBRO - 1991

Este exemplar corresponde à redação final da defendida por Rodrigo Martinez Spanó e aprovada pela Comi Julgadora em 18/12/91.

Orientador

*À meus pais e irmãos*

## AGRADECIMENTOS

Ao Prof. Dr. Maurício Ferreira Magalhães, pela orientação, dedicação, incentivo e amizade que me dispensou durante o desenvolvimento deste trabalho.

Ao CNPq, cujo apoio financeiro permitiu a realização deste trabalho.

Ao amigo Alencar, pela implementação dos escalonadores inseridos na ferramenta SSE e pela valiosa colaboração na fase de testes do sistema.

Aos amigos de "HRT", Sibelius, Juan e Adilson pelas importantes discussões e sugestões.

À Celso, Armando e Gustavo, cuja amizade e convívio diário trouxeram muita força.

À Hsieh e Paulo Minoru, pela grande amizade desde a época de São Carlos.

Aos amigos Paulo Maurício, Conceição, Humberto, Ruy e Telma, pelo constante incentivo e momentos de felicidade que compartilhamos no dia à dia.

Aos pessoal da MS Imagem, Marcelo Souza, Fátima, Wilson, Daniel, "Carioca", Kathia e Renato, pelo apoio e amizade.

Aos primos Eugênio e "Zezé", pela calorosa acolhida em Campinas.

Aos amigos Paulo, José Sergio, Marcia, Maria Alice e "Ciça" que, apesar da distância, serão sempre lembrados.

# SUMÁRIO

# SUMÁRIO

# LISTA DE FIGURAS

# RESUMO

# RESUMO

A teoria de escalonamento representa uma importante área de pesquisa nos sistemas de tempo real crítico. Este trabalho propõe uma ferramenta de software, denominada *Sistema de Simulação para Escalonadores* (*SSE*), dedicada ao estudo de algoritmos de escalonamento. O SSE possui dois objetivos principais:

(1) - permitir a análise da execução (simulada) de um conjunto de tarefas de uma aplicação de tempo real crítico, diante de um algoritmo de escalonamento escolhido. Para este caso, o SSE fornece ao usuário, através de uma interface amigável, várias facilidades;

(2) - servir de ambiente à implementação de algoritmos de escalonamento. Através de uma interface de software bem definida, esta opção do SSE possibilita ao usuário implementar políticas de escalonamento e integrá-las ao sistema para estudo posterior, o que torna a ferramenta bastante flexível.

## ABSTRACT

The scheduling theory is an important research field for hard real-time systems. This work presents a software tool, named Sistema de Simulação para Escalonadores - SSE (Simulation System for Schedulers), oriented towards scheduling algorithms study. The SSE has two main goals:

(1) - allows the simulated run-time analyse of a task set of a hard real-time application, using a particular scheduling algorithm. In this case, the SSE provides to the user many facilities, through a friendly user interface.

(2) - provides an environment for scheduling algorithms implementation. Using a well defined software interface, this SSE's option allows an user to implement new scheduling policies and integrate them to the system for following study, what makes the tool very flexible.

# CAPÍTULO I

# INTRODUÇÃO

# CAPÍTULO I

## INTRODUÇÃO

O interesse pelos sistemas de tempo real tem crescido devido à grande quantidade de aplicações surgidas nas mais diversas áreas. A complexidade destes sistemas varia de simples processamentos a tarefas altamente sofisticadas.

Os sistemas de tempo real podem ser divididos em duas classes: *sistemas de tempo real não-crítico* ("soft real-time systems") e *sistemas de tempo real crítico* ("hard real-time systems").

Na primeira categoria (tempo real não-crítico), as tarefas que fazem parte do sistema não possuem prazos para terminar de executar, sendo que o principal objetivo é minimizar o tempo de resposta das tarefas. Já nos sistemas de tempo real crítico, o funcionamento correto da aplicação depende, não apenas dos resultados lógicos, mas também, do instante em que eles são produzidos. As tarefas que fazem parte deste tipo de sistema possuem restrições de tempo ("ready time" e "deadline") que devem ser cumpridas. A violação dos requisitos temporais destas tarefas pode trazer consequências catastróficas, conforme a aplicação em questão.

Exemplos de aplicações de tempo real crítico são: controle de vôo para aviões e naves espaciais, sistemas de manufatura , controle em usinas nucleares, controle de experimentos em laboratórios, exploração submarina, robótica, etc. Para os sistemas de tempo real não-crítico, que atualmente são os mais encontrados, pode-se citar como exemplo de aplicações: sistemas de reserva de passagem nas companhias aéreas, transações em banco de dados [STA1 88], sistemas de automação bancária, entre outros.

Devido a sua importância e complexidade, os sistemas de tempo real crítico têm despertado interesse em várias áreas da computação. As principais áreas de pesquisa são: *especificação e verificação de sistemas, teoria de escalonamento, sistemas operacionais, metodologias de projeto, linguagens de programação, tolerância a falhas, arquitetura de computadores, sistemas distribuídos e inteligência artificial.* Estas áreas procuram fornecer teorias e tecnologias que permitam o desenvolvimento de sistemas de tempo real levando em consideração seus requisitos temporais. Apesar de

ainda serem poucos os resultados, já existem trabalhos interessantes como é o caso do projeto SPRING [RAM 89].

Entre as áreas de pesquisa citadas, a teoria de escalonamento é uma das que tem recebido mais atenção. Nesta área são estudados algoritmos de escalonamento, os quais desempenham importante papel no cumprimento das restrições de tempo das aplicações. As linhas de trabalho adotadas para o estudo da teoria de escalonamento são: teoria dos grafos, programação matemática e heurística.

Existem algoritmos clássicos de escalonamento para tempo real crítico, como por exemplo o "Earliest Deadline" e o Taxa Monotônica [LIU 73], e que, comprovados através de resultados formais, são considerados ótimos. Porém, estes algoritmos assumem suposições em relação às tarefas e/ou ambiente em que são executadas, que restringem bastante as aplicações reais onde podem ser usados. Novas estratégias de escalonamento vêm sendo propostas na tentativa de tratar os requisitos encontrados nas aplicações de tempo real crítico. Dependendo da complexidade acrescentada por estes requisitos, o problema de escalonamento pode se tornar NP-hard ou NP-completo, sendo necessário trabalhar com soluções heurísticas para resolvê-lo. Neste caso, para se conhecer a eficiência do algoritmo de escalonamento, são necessários vários testes.

Por outro lado, a troca da política de escalonamento presente em um núcleo de tempo real quando possível (supondo-se ter acesso ao código fonte), requer adaptações em geral não triviais, não sendo rara a necessidade de projetá-lo novamente. Este fato representa um obstáculo ao teste de novos algoritmos de escalonamento, utilizando-se um núcleo de tempo real.

Buscando trazer maior facilidade ao estudo de algoritmos de escalonamento para sistemas de tempo real crítico, o presente trabalho propõe uma ferramenta de software, denominada *Sistema de Simulação para Escalonadores* (*SSE*), cujas finalidades são: (1) permitir a análise de políticas de escalonamento, observando a execução simulada de um conjunto de tarefas especificadas pelo usuário e (2) servir de ambiente para a implementação de novas políticas de escalonamento. Na primeira opção, existe um conjunto de escalonadores prontos para serem utilizados, bastando para isto que seja selecionado um deles. A segunda opção garante maior flexibilidade à ferramenta, possibilitando que novos algoritmos de escalonamento sejam implementados e analisados, sendo necessário, neste caso, que se obedeça uma interface de software bem definida.

O trabalho está organizado do seguinte modo: o capítulo II apresenta conceitos importantes da teoria de escalonamento aplicada aos sistemas de tempo real crítico. Também é feita uma revisão de alguns algoritmos conhecidos da literatura. O capítulo III descreve a ferramenta de simulação SSE, mostrando todas as suas características de modo a atender às duas finalidades para as quais foi projetada. O capítulo IV discute os aspectos mais relevantes da implementação do SSE. Finalizando, o capítulo V traz as conclusões do trabalho e sugestões para a sua continuidade.

# Capítulo II

## Algoritmos de Escalonamento

## para

## Sistemas de Tempo Real Crítico

# CAPÍTULO II

# ALGORITMOS DE ESCALONAMENTO PARA SISTEMAS DE TEMPO REAL CRÍTICO

## 2.1 - Definição de Escalonadores

Nos sistemas de tempo real existem várias tarefas concorrentes. A escolha de qual tarefa obterá determinado recurso (por exemplo o processador) em um dado instante, é feita por um módulo do sistema operacional ou núcleo de tempo real denominado *escalonador*. Este módulo é a implementação de um *algoritmo de escalonamento* que, baseado nas características das tarefas e do ambiente em que são executadas, atribui-lhes prioridade para ocupar o recurso negociado.

Os sistemas de tempo real crítico são caracterizados por conter tarefas que possuem restrições de tempo além de, possivelmente, restrições de precedência e compartilhamento de recursos. Tais sistemas não podem ter suas restrições temporais violadas, senão correm o risco de causar graves acidentes. Assim, os algoritmos de escalonamento representam papel importante nestes sistemas, pois uma ordem inadequada na execução das tarefas pode provocar atrasos desnecessários e, consequentemente, a violação dos requisitos temporais da aplicação.

Em função das diversas características que um sistema de tempo real crítico pode ter, existe uma grande variedade de algoritmos de escalonamento, onde cada um busca tratar aspectos particulares do sistema. Neste sentido, é conveniente dividí-los em grupos segundo certos critérios.

## 2.2 - Classificação

Os algoritmos de escalonamento de tempo real crítico podem ser divididos em duas classes: *estáticos* e *dinâmicos* [CHE 87]. Os algoritmos estáticos calculam todo o escalonamento para as tarefas, atribuindo-lhes prioridades fixas, antes que elas comecem a executar ("off-line"). Tal tipo de algoritmo requer a priori um conhecimento completo das características das tarefas que fazem parte do sistema. Já os algoritmos

dinâmicos determinam o escalonamento das tarefas conforme a evolução da execução do sistema, atribuindo-lhes prioridades que podem mudar com o tempo.

Comparando os dois tipos de algoritmos conclui-se que, apesar dos estáticos terem baixo "overhead", eles são inflexíveis e não conseguem se adaptar às alterações do ambiente, sendo inapropriados a aplicações, cujo comportamento não seja previsível. Por outro lado, os algoritmos do tipo dinâmico apresentam um maior "overhead" devido à constante mudança de prioridade das tarefas mas, em compensação, fornecem maior flexibilidade ao sistema, podendo facilmente se adaptar às mudanças do ambiente.

Os algoritmos de escalonamento também podem ser classificados com relação à arquitetura do ambiente em que serão executados: *centralizados* ou *distribuídos.* Algoritmos de escalonamento centralizados são aqueles empregados em sistemas monoprocessadores ou multiprocessadores onde o custo de comunicação entre processadores é insignificante. Os algoritmos de escalonamento distribuídos estão relacionados a sistemas com vários nodos processadores interconectados, cujo custo de comunicação entre eles não é desprezível. Nos sistemas distribuídos, o custo de comunicação entre processadores é um fator importante que deve ser levado em consideração no escalonamento de tarefas.

A figura 2.1 mostra a classificação geral dos algoritmos de escalonamento de tempo real crítico.

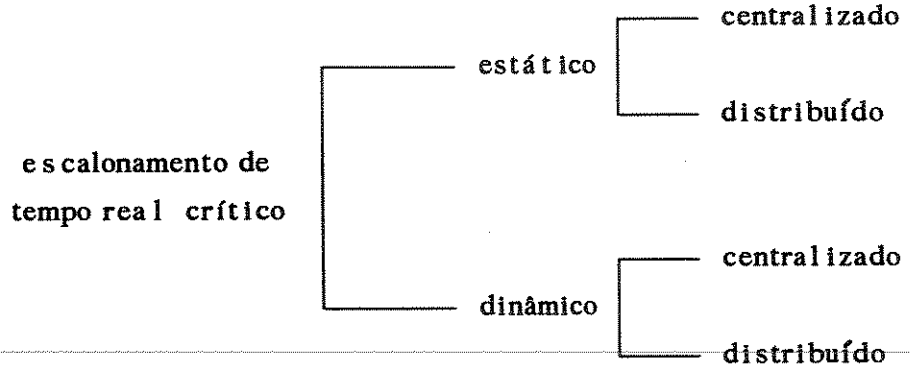



Figura 2.1 - Classificação dos algoritmos de escalonamento de tempo real crítico

## 2.3 - Características das Tarefas

A tarefa é um módulo de software que pode ser invocado para executar uma função específica e que, em conjunto com as demais tarefas do sistema, contribui para o processamento global da aplicação. A tarefa é a entidade escalonada dentro do sistema. Em sistemas estáticos, o número de tarefas que serão executadas é conhecido a priori e não se altera. Já para sistemas dinâmicos, é possível ocorrer a chegada de novas tarefas em qualquer instante de execução, assim, o número de tarefas que devem ser escalonadas pode variar no decorrer do tempo.

As tarefas em um sistema de tempo real crítico são caracterizadas sob vários aspectos, como será visto a seguir.

### (a) - *Tempo*

As tarefas possuem alguns parâmetros relacionados ao tempo, a saber:

- *tempo de chegada* ("arrival time") (A): é o instante em que a tarefa é invocada no sistema;

- *tempo de pronto* ("ready time") (R): é o instante mais cedo que uma tarefa pode iniciar sua execução. Observa-se que $R \geq A$. Uma simplicação adotada em vários algoritmos é assumir $R = A$;

- *tempo de execução* ("computation time") (C): é o tempo máximo que uma tarefa pode levar para executar. No cálculo do valor deste parâmetro, pode ser levado em conta também "overheads" introduzidos pelo s.o./núcleo de tempo real como, por exemplo, o tempo para mudança de contexto na troca de tarefas.

- *deadline* (D): é o prazo que uma tarefa tem para terminar de executar. As tarefas, cuja a execução não pode ultrapassar seu prazo, são ditas ter "hard deadline". Já as tarefas onde a violação do seu prazo é aceitável são ditas ter "soft deadline".D é um valor relativo ao tempo de chegada (A) da tarefa.

(b) - *Periodicidade*

Esta característica diz respeito à frequência com que a tarefa é executada. As tarefas podem ser *aperiódicas* ou *periódicas*.

Tarefas aperiódicas são aquelas que podem ser invocadas a qualquer instante. Módulos para tratamento de falhas de um sistema são exemplos deste tipo de tarefa.

As tarefas periódicas são aquelas invocadas apenas uma vez por um período P. Como exemplo deste tipo de tarefa, pode-se citar tarefas que realizam o monitoramento de sensores. O tempo de chegada e o "deadline" das instâncias de uma tarefa com período P são dados por:

$$A(T_{i+1}) = A(T_i) + P$$
$$A(T_i) + C \leq D(T_i) \leq A(T_{i+1})$$

onde

C = tempo de execução da tarefa T
$A(T_i)$ = tempo de chegada da instância i da tarefa T
$D(T_i)$ = "deadline" absoluto da instância i da tarefa T ($A(T_i)$ + D).

(c) - *Precedência*

As tarefas podem ser *independentes* entre si ou terem *restrições de precedência*. No primeiro caso, não importa a ordem em que as tarefas são executadas. Quando há restrições de precedência entre as tarefas, existe especificado um relacionamento entre elas. Uma tarefa $T_i$ é dita preceder uma tarefa $T_j$, se $T_i$ deve terminar antes que $T_j$ inicie.

(d) - *Comunicação*

As tarefas podem ser *independentes* ou *comunicantes* entre si para obter sincronização, assim como, para trocar dados.

(e) - *Recursos*

As tarefas podem necessitar de recursos específicos para executar como, por exemplo: memória, periféricos de I/O, processadores específicos, entre outros.

(f) - *Preempção*

Esta característica diz respeito à continuidade de execução das tarefas. Uma tarefa *preemptível* permite que sua execução seja interrompida por outra tarefa de maior prioridade em um instante qualquer, voltando posteriormente ao seu processamento, no ponto em que havia parado. Já uma tarefa *não-preemptível*, quando tem iniciado seu processamento, deve executar sem interrupção até o instante em que resolva liberar o processador.

Devido à variedade de configuração dos ambientes de tempo real, assim como às características das tarefas, os algoritmos de escalonamento são desenvolvidos procurando tratar aspectos específicos. Não existem algoritmos genéricos que englobem todas as situações e sejam operacionais (baixo "overhead").

Na próxima seção, serão discutidos algoritmos clássicos de escalonamento para sistemas de tempo real crítico.

## 2.4 - Algoritmos de Escalonamento

Os algoritmos de escalonamento apresentados a seguir são para ambiente centralizado (monoprocessador), sendo esta a configuração a que o SSE é dedicado. Assume-se que o tempo de pronto para as tarefas é igual ao tempo de chegada (R = A).

### 2.4.1 - Taxa Monotônica

Este algoritmo, desenvolvido por Liu e Layland [LIU 73], é do tipo estático (prioridades fixas), para tarefas periódicas, independentes (sem precedência ou comunicação), preemptíveis e com prazo ("deadline") de execução.

Dado um conjunto de tarefas, o algoritmo *taxa monotônica* atribui a cada tarefa uma prioridade fixa conforme os respectivos períodos. Quanto menor for o período da tarefa, maior será sua prioridade e vice-versa.

O exemplo a seguir ilustra o algoritmo. Considere duas tarefas T1 e T2 com os parâmetros dados na tabela 2.1. A tarefa T1 tem prioridade maior que T2, pois seu período é menor. Por convenção, assume-se que quanto menor o valor, maior será a prioridade. A figura 2.2 mostra um diagrama de Gantt que representa a execução das tarefas no transcorrer do tempo.

| Tarefa | Tempo de Execução | Período | Prioridade |
|---|---|---|---|
| T1 | 4 | 10 | 1 |
| T2 | 8 | 20 | 2 |

Tabela 2.1 - Especificações das tarefas

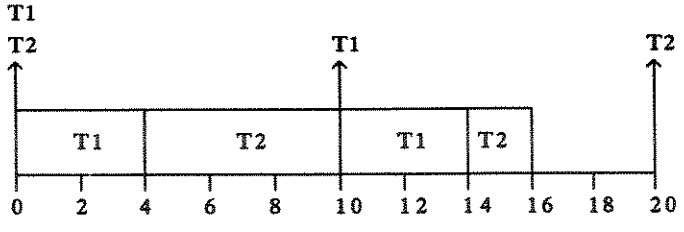



Figura 2.2 - Execução do algoritmo *taxa monotônica*

No instante 0 a tarefa T1 começa a executar. Ao terminar no instante 4, T2 assume a CPU e permanece até o instante 10 quando é interrompida devido à invocação da segunda instância de T1 que novamente executa até o instante 14, quando então dá lugar a T2 para finalizar seu processamento pendente.

Liu e Layland determinaram uma condição suficiente para garantir a não violação dos prazos de um conjunto de tarefas periódicas. O resultado é dado através do teorema a seguir [LIU 73].

*Teorema 1*: Um conjunto de n tarefas periódicas independentes, escalonadas pelo algoritmo *taxa monotônica*, sempre respeitará o "deadline" de suas tarefas se

$$U(n) = \sum_{i=1}^{n} C_i/P_i = C_1/P_1 + \ldots + C_n/P_n \leq n(2^{1/n} - 1) = LU(n)$$

onde

$C_i$ = tempo de execução da tarefa $T_i$

$P_i$ = período da tarefa $T_i$

$U(n) = \sum_{i=1}^{n} C_i/P_i$ é o *fator de utilização do processador* para o conjunto de tarefas. LU(n) representa o *limite para o fator de utilização do processador*. Este limite converge para 69% (ln 2) conforme o número de tarefas tende para infinito.

Para as tarefas do exemplo anterior (tabela 2.1), o valor do fator de utilização do processador é dado por:

$$\sum_{i=1}^{2} C_i/P_i = C_1/P_1 + C_2/P_2 = 4/10 + 8/20 = 4/5 = 0,8$$

O limite do fator de utilização é:

$$LU(2) = 2(2^{1/2} - 1) = 0,828$$

Como $\sum_{i=1}^{2} C_i/P_i \leq LU(2)$, o conjunto de tarefas com certeza tem seus "deadlines" garantidos.

Na prática, verifica-se que o limite para o fator de utilização apresentado no teorema 1 é um valor pessimista, pois nem sempre um conjunto de tarefas tem a execução para o pior caso (quando todas as tarefas possuem o mesmo tempo de pronto). Lehoczky, Sha e Ding [LEH 89] fizeram uma análise estocástica para um conjunto de tarefas geradas aleatoriamente, escalonadas pelo algoritmo *taxa monotônica* e concluiram que uma boa aproximação para o limite de escalonabilidade é 88%.

Quando o fator de utilização do processador para um conjunto de tarefas está acima do limite especificado pelo teorema 1, não significa que as tarefas não cumprirão seus respectivos prazos. Sha e Goodenough desenvolveram para este caso um teste de escalonabilidade exato [SHA 90].

O algoritmo *taxa monotônica* tem sido bastante estudado em vista de sua eficiência. Novas propostas trazendo variações para este algoritmo têm surgido.

### 2.4.2 - Taxa Monotônica com Tarefas Aperiódicas

O algoritmo *taxa monotônica* proposto por Liu e Layland trabalha apenas com tarefas periódicas. Para tratar invocações de tarefas aperiódicas existem alguns algoritmos cujos objetivos são: garantir a execução dentro dos prazos para as tarefas aperiódicas com "hard deadline" e minimizar o tempo de resposta médio das tarefas aperiódicas com "soft deadline". O cumprimento destes objetivos não deve comprometer os prazos das tarefas periódicas.

#### 2.4.2.1 - "Background Service"

Neste algoritmo as tarefas periódicas seguem a política de *taxa monotônica*, enquanto que as tarefas aperiódicas são executadas apenas quando o processador estiver inativo (isto é, quando não há tarefas periódicas executando ou pendentes).

Considere como exemplo quatro tarefas $T_1$, $T_2$, $T_3$ e $T_4$ com os parâmetros dados nas tabelas 2.2 e 2.3. $T_1$ e $T_2$ são tarefas periódicas e $T_3$ e $T_4$ aperiódicas. A figura 2.3 mostra o diagrama de Gantt para a execução deste conjunto de tarefas entre os intantes 0 e 20 unidades de tempo.

| Tarefas Periódicas | Tempo de Execução | Período | Prioridade |
|---|---|---|---|
| T1 | 4 | 10 | 1 |
| T2 | 8 | 20 | 2 |

Tabela 2.2 - Especificação das tarefas periódicas

| Tarefas Aperiódicas | Tempo de Execução | Chegada |
|---|---|---|
| T3 | 1 | 6 |
| T4 | 1 | 12 |

Tabela 2.3 - Especificação das tarefas aperiódicas

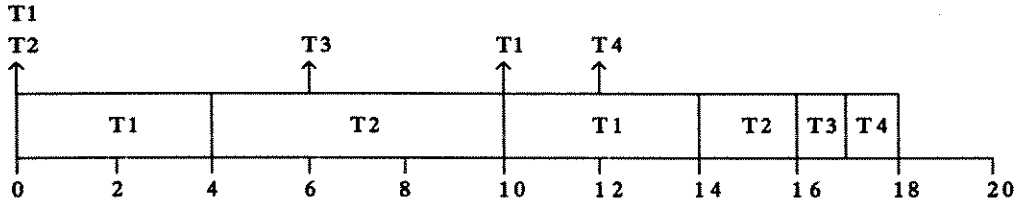



Figura 2.3 - Execução do algoritmo *"background service"*

Um problema no algoritmo *"background service"* é que se a utilização por parte das tarefas periódicas for alta, a oportunidade de execução dada às tarefas aperiódicas será pequena, o que acarretará aumento no tempo de resposta destas tarefas.

#### 2.4.2.2 - "Polling Server"

Este algoritmo cria uma tarefa periódica, denominada "polling", para servir às invocações de tarefas aperiódicas. Em intervalos regulares, a tarefa "polling" é invocada e atende os pedidos pendentes de tarefas aperiódicas. Caso não haja alguma invocação, a tarefa "polling" suspende a si própria até seu próximo período, e o tempo alocado originalmente para servir tarefas aperiódicas é utilizado pelas tarefas

periódicas. Sendo a tarefa "polling" periódica, sua prioridade é dada pela política de *taxa monotônica*, assim como para as demais tarefas periódicas do sistema.

Um exemplo, usando o algoritmo *"polling server"*, é dado a seguir. As características das tarefas encontram-se nas tabela 2.4 e 2.5. Foi criada uma tarefa "polling" (P) com tempo de execução 1 e período 5, com prioridade 1 (pois tem o menor período). A execução das tarefas pode ser vista na figura 2.4.

| Tarefas Periódicas | Tempo de Execução | Período | Prioridade |
|---|---|---|---|
| T1 | 4 | 10 | 2 |
| T2 | 8 | 20 | 3 |
| P | 1 | 5 | 1 |

Tabela 2.4 - Especificação das tarefas periódicas

| Tarefas Aperiódicas | Tempo de Execução | Chegada |
|---|---|---|
| T3 | 1 | 5 |
| T4 | 0.5 | 12 |

Tabela 2.5 - Especificação das tarefas aperiódicas

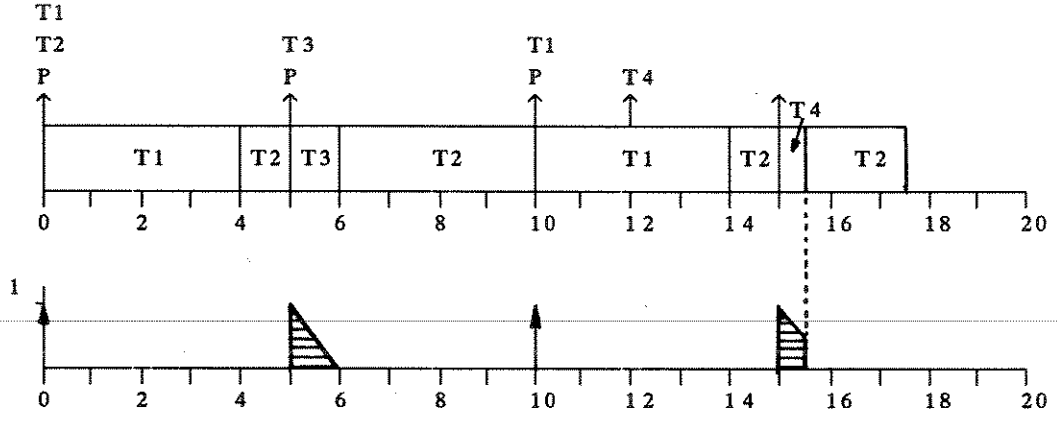



Figura 2.4 - Execução do algoritmo *"polling service"* e gráfico de capacidade de serviço.

O gráfico abaixo do diagrama de Gantt na figura 2.4 refere-se à *capacidade de serviço* da tarefa "polling". Esta medida significa o tempo de execução que a tarefa "polling" pode oferecer às tarefas aperiódicas num dado instante. Observa-se no exemplo que, nos instantes 0 e 10, a capacidade de serviço vai ao máximo (1), mas como não havia invocações pendentes de tarefas aperiódicas, seu valor cai a zero. Se a invocação de uma tarefa aperiódica ocorrer logo após a tarefa "polling" tiver sido suspensa, ela deverá esperar o início do próximo período desta, ou algum instante em que o processador estiver inativo (*"background service"*).

O tempo de resposta para as tarefas aperiódicas depende do período e da capacidade de serviço da tarefa "polling".

Apesar do algoritmo *"polling server"* oferecer melhor tempo de resposta para tarefas aperiódicas do que o *"background service"*, nem sempre ele pode servir a uma invocação imediatamente. Lehoczky, Sha, Strosnider introduziram os algoritmos "Deferrable Server" (DS) e "Priority Exchange" (PE) [LEH 87] que superam os problemas encontrados nos dois algoritmos anteriores.

Os algoritmos DS e PE criam uma tarefa periódica para servir às invocações de tarefas aperiódicas. Ao contrário do *"polling server"*, estes algoritmos preservam o tempo de execução alocado para servir tarefas aperiódicas, mesmo quando não há invocações pendentes. Os algoritmos PE e DS diferem apenas na maneira pela qual eles preservam o tempo de execução da tarefa servidora. Os algoritmos DS e PE serão apresentados a seguir.

#### 2.4.2.3 - "Deferrable Server" (DS)

Como o algoritmo *"polling server"*, o *"deferrable server"* cria uma tarefa periódica para servir às tarefas aperiódicas. Entretanto, este algoritmo preserva o tempo de execução alocado às tarefas aperiódicas após a invocação da tarefa servidora.

O DS mantém o tempo de execução , reservado às tarefas aperiódicas, durante o período da tarefa servidora. Assim, as invocações aperiódicas podem ser tratadas ao nível da prioridade da tarefa servidora a qualquer instante, enquanto a capacidade de serviço não tiver sido esgotada.

No início do período da tarefa servidora, o tempo de execução para as tarefas aperiódicas é completado novamente até sua capacidade máxima (tempo de execução da tarefa servidora).

A figura 2.5 mostra um exemplo trabalhando-se com o algoritmo DS. As características das tarefas é dada pelas tabelas 2.6 e 2.7.

S = Tarefa Servidora

| Tarefas Periódicas | Tempo de Execução | Período | Prioridade |
|---|---|---|---|
| T1 | 4 | 10 | 2 |
| T2 | 8 | 20 | 3 |
| S | 1 | 5 | 1 |

Tabela 2.6 - Especificação das tarefas periódicas

| Tarefas Aperiódicas | Tempo de Execução | Chegada |
|---|---|---|
| T3 | 1 | 5 |
| T4 | 0.5 | 12 |

Tabela 2.7 - Especificação das tarefas aperiódicas

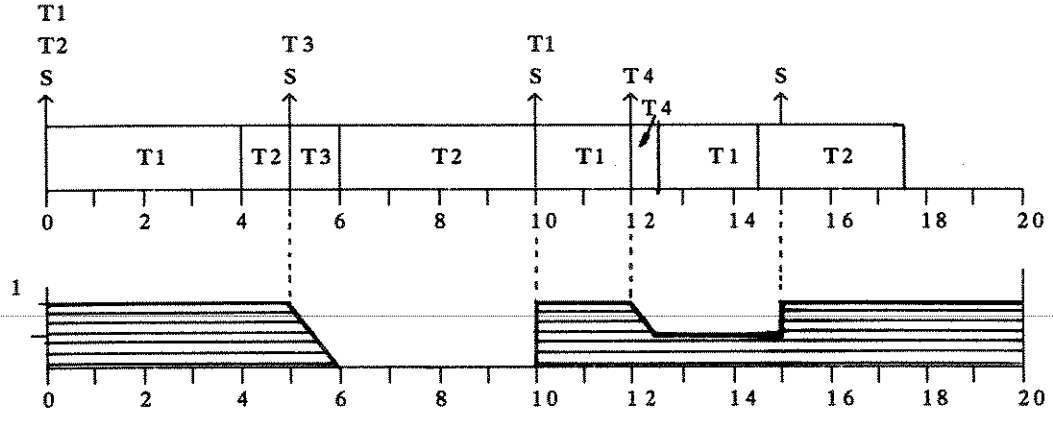



Figura 2.5 - Execução do algoritmo *"deferrable server"* e gráfico de capacidade de serviço

Neste exemplo, é criada uma tarefa servidora com alta prioridade (atribuindo-lhe o menor período entre as demais tarefas), com tempo de execução 1. Seguindo o gráfico na figura 2.5, pode-se observar que a capacidade de serviço que a tarefa servidora dispõe inicialmente, encontra-se no valor máximo (tempo de execução = 1), sendo mantida até a ocorrência da primeira invocação de uma tarefa aperiódica ($T_3$) no instante 5. Neste momento, a tarefa $T_2$ é interrompida e $T_3$ é tratada imediatamente, esgotando a capacidade de serviço da tarefa servidora no instante 6. Em t = 10 a tarefa servidora recupera sua capacidade de serviço, pois é o início de um novo período. No instante 12, uma segunda tarefa aperiódica chega e é atendida. Como esta tarefa só necessita de 0.5 do tempo de execução da tarefa servidora, o restante da capacidade é preservada. Desta forma, o algoritmo DS fornece melhor tempo de resposta do que o algoritmo *"polling server"*, pois ele mantém seu tempo de execução enquanto este não é usado totalmente.

#### 2.4.2.4 - "Priority Exchange" (PE)

O algoritmo PE preserva o tempo de execução reservado à tarefa servidora, trocando-o pelo tempo de execução de uma tarefa periódica de prioridade menor que a sua. No início do período da tarefa servidora o seu tempo de execução recebe um valor máximo pré-determinado. Se existir alguma tarefa aperiódica pendente, esta é atendida no nível de prioridade da tarefa servidora. Caso contrário, escolhe-se a tarefa periódica de maior prioridade para executar e ocorre uma troca de prioridade. Neste caso, a tarefa periódica executa no nível de prioridade da tarefa servidora, e o tempo de execução reservado à tarefa servidora é acumulado no nível de prioridade da tarefa periódica. As trocas de prioridades continuam até que o tempo de execução da tarefa servidora seja utilizado completamente.

Sendo o objetivo do algoritmo PE fornecer tempo médio de resposta baixo para as tarefas aperiódicas, quando ocorrer empate de prioridade entre invocações, as tarefas aperiódicas devem sempre ser favorecidas.

O exemplo a seguir ilustra o funcionamento do algoritmo PE. As tarefas são dadas pelas tabelas 2.8 e 2.9. A figura 2.6 mostra a evolução do escalonamento das tarefas junto com o gráfico de capacidade de serviço da tarefa servidora. Como o algoritmo PE deve gerenciar o tempo de execução que a tarefa servidora tem disponível

nos vários níveis de prioridade do sistema, a figura 2.6 apresenta três gráficos de capacidade de serviço x tempo, um para cada nível de prioridade. O nível 1 corresponde à prioridade mais alta, dada à tarefa servidora, seguido pelos níveis 2 e 3, representando a prioridade das tarefas periódicas $T_1$ e $T_2$, respectivamente.

S = Tarefa Servidora

| Tarefas Periódicas | Tempo de Execução | Período | Prioridade |
|---|---|---|---|
| $T_1$ | 4 | 10 | 2 |
| $T_2$ | 8 | 20 | 3 |
| S | 1 | 5 | 1 |

Tabela 2.8 - Especificação das tarefas periódicas

| Tarefas Aperiódicas | Tempo de Execução | Chegada |
|---|---|---|
| $T_3$ | 1 | 5 |
| $T_4$ | 0.5 | 12 |

Tabela 2.9 - Especificação das tarefas aperiódicas

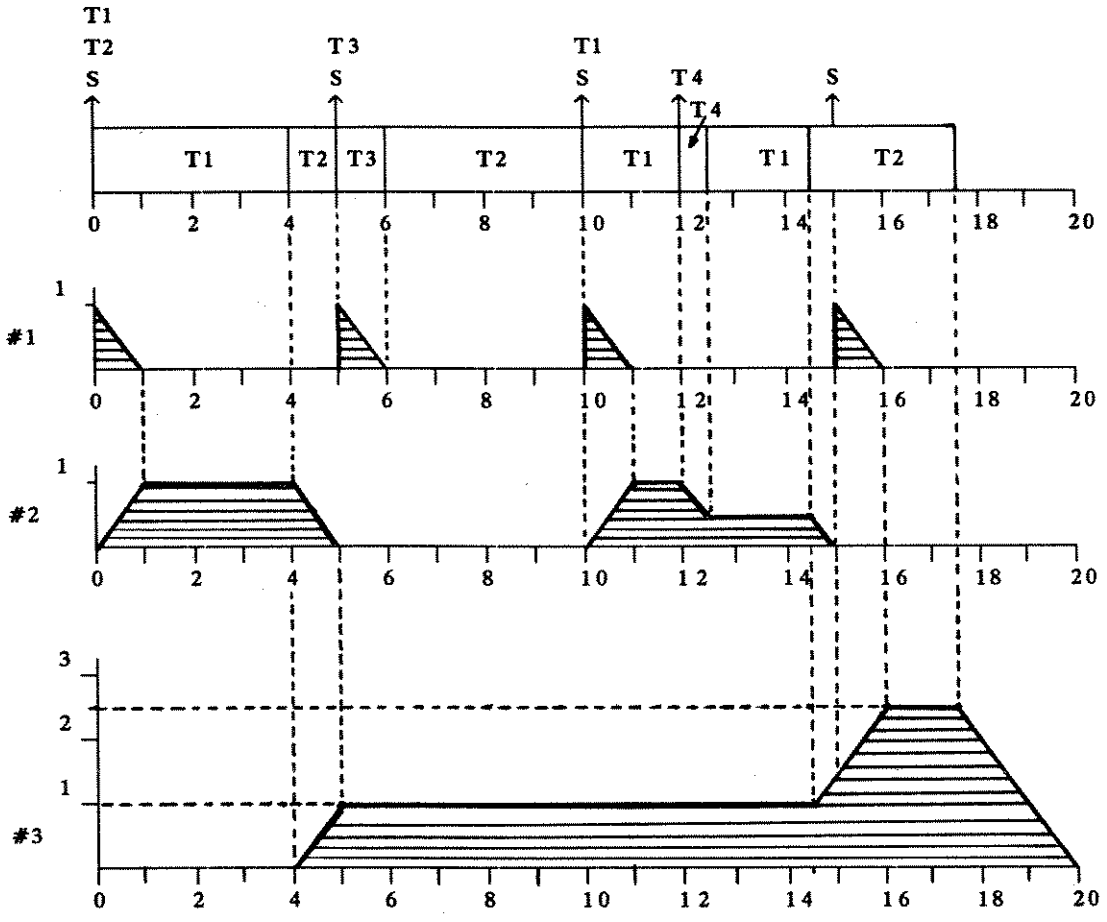



Figura 2.6 – Execução do algoritmo *"priority exchange"* e gráfico de capacidade de serviço

No início, a capacidade da tarefa servidora tem seu valor máximo, mas como não há tarefas aperiódicas pendentes, ocorre uma troca de prioridades entre os níveis 1 e 2. A tarefa servidora ganha capacidade de serviço no nível de prioridade 2 e a tarefa periódica $T_1$ executa com prioridade 1. No instante 4, a tarefa $T_1$ completa e $T_2$ inicia. Como não há tarefas aperiódicas pendentes, uma outra troca de prioridades ocorre entre os níveis 2 e 3. No instante 5, o tempo de execução da tarefa servidora no nível de prioridade 1 é completado até seu valor máximo e é utilizado para fornecer serviço imediato à primeira invocação de tarefa aperiódica. No instante 10, o tempo de execução da tarefa servidora na prioridade 1 recebe seu valor máximo e é

trocado com o nível 2 de prioridade. No instante 12, o tempo de execução no nível 2 é usado para servir à segunda invocação aperiódica. No instante 14.5, o tempo de execução restante na prioridade 2 é trocado com a prioridade 3. No instante 15, a nova capacidade de serviço recebida no nível 1 é trocada com o nível 3 de prioridade. Finalizando, no instante 17.5, o tempo de execução na prioridade 3 é descartado, pois não há tarefas periódicas ou aperiódicas pendentes.

Comparando os algoritmos DS e PE, verifica-se que cada um possui seus méritos. A vantagem do algoritmo DS sobre o PE é que ele é mais simples conceitualmente, levando assim a uma implementação mais fácil. Entretanto, Lehoczky, Sha e Strosnider [LEH 89] chegaram a resultados matemáticos que provam que o limite de escalonamento periódico (maior utilização para a qual o algoritmo *taxa monotônica* garante escalonar um conjunto de tarefas periódicas) é maior para o algoritmo PE. Para um valor de utilização de tarefas periódicas, observou-se também que o tamanho da tarefa servidora (tempo de execução reservado à ela) que o PE pode ter é maior que o do DS.

Sprunt, Sha e Lehoczky [SPR 89] propuseram um algoritmo, para tarefas aperiódicas com "soft deadline", denominado *"sporadicserver"* (SS) que engloba as vantagens dos algoritmos DS e PE e exclui suas limitações.

#### 2.4.2.5 - "Sporadic Server" (SS)

O algoritmo SS, como os algoritmos DS e PE, cria uma tarefa de alta prioridade para servir as tarefas aperiódicas. O SS preserva o tempo de execução da tarefa servidora, que se encontra em um nível alto de prioridade, até que a invocação de uma tarefa aperiódica ocorra. O algoritmo SS difere dos algoritmos DS e PE no modo em que ele retorna para a tarefa servidora o tempo gasto pelas tarefas aperiódicas. Os algoritmos DS e PE retornam periodicamente este tempo enquanto que, o algoritmo SS apenas fornece tempo de execução para a tarefa servidora se houve algum consumo por uma tarefa aperiódica. O modo o qual é feito este fornecimento de tempo de execução é explicado a seguir, mas antes são definidos alguns termos utilizados.

Ps: nível de prioridade da tarefa, a qual o sistema está executando;

Pi: um dos níveis de prioridade. $P_1 > P_2 > \ldots > P_n$;

**Ativo:** Um nível de prioridade Pi é considerado ativo, se a prioridade atual do sistema (Ps) é igual ou maior que a prioridade Pi;
**Inativo:** um nível de prioridade Pi é inativo, se a prioridade atual do sistema Ps é menor que a prioridade Pi;
**RTi:** é o instante no qual o tempo de execução consumido da tarefa servidora de prioridade Pi será devolvido a ela.

A devolução do tempo consumido da tarefa servidora, cuja prioridade é Pi, procede da seguinte maneira:

Se a tarefa servidora tem tempo de execução disponível, o instante de devolução (RTi) é atribuído quando o nível de prioridade Pi se torna ativo. O valor de RTi é igual ao instante atual mais o período da tarefa servidora naquela prioridade. Se a capacidade de serviço foi esgotada, o próximo instante de devolução pode ser atribuido, quando esta capacidade se tornar maior que zero e Pi estiver ativo.

A devolução de algum tempo de execução consumido para a tarefa servidora deverá ocorrer em RTi, se o nível de prioridade Pi se tornar inativo ou o tempo de execução da tarefa servidora for esgotado. A quantidade a ser devolvida é igual à quantidade de tempo de execução da tarefa servidora consumida.

A seguir, é dado um exemplo que ilustrará a operação do algoritmo SS. As tarefas são dadas pelas tabelas 2.10 e 2.11. Observa-se que o maior tempo de execução da tarefa servidora foi determinado pelos resultados encontrados em [SPR 89]. Pela política de *taxa monotônica*, a ordem das tarefas periódicas por prioridades é S > $T_1$ > $T_2$

S = Tarefa Servidora

| Tarefas Periódicas | Tempo de Execução | Período | Prioridade |
|---|---|---|---|
| $T_1$ | 2 | 10 | 2 |
| $T_2$ | 6 | 14 | 3 |
| S | 1.33 | 5 | 1 |

Tabela 2.10 - Especificação das tarefas periódicas

| Tarefas Aperiódicas | Tempo de Execução | Chegada |
|---|---|---|
| T3 | 1.33 | 1 |
| T4 | 1.33 | 8 |

Tabela 2.11 - Especificação das tarefas aperiódicas

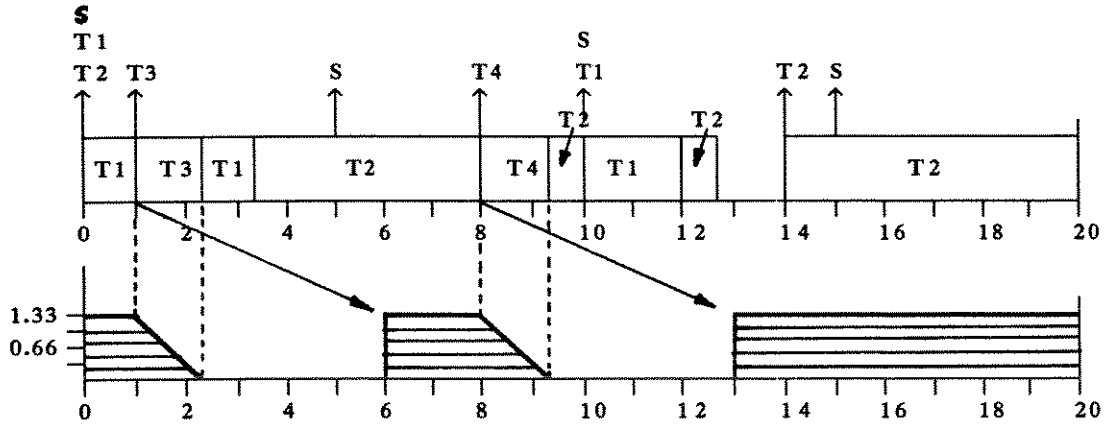



Figura 2.7 - Execução do algoritmo *"sporadic server"* e gráfico de capacidade de serviço

A figura 2.7 mostra o comportamento do algoritmo SS para o conjunto de tarefas dadas.

Como a tarefa servidora é a única tarefa executando no nível de prioridade $P_1$ (maior prioridade), $P_1$ se torna ativa apenas quando a tarefa servidora executa uma tarefa aperiódica. Assim, $RT_1$ é determinado sempre que uma tarefa aperiódica for executada. A devolução do tempo de execução consumido ocorrerá um período (da tarefa servidora) após o instante, no qual a tarefa servidora atende uma tarefa aperiódica.

No início, a tarefa servidora possui a sua capacidade máxima de serviço. Em $t = 1$, ocorre a primeira invocação de tarefa aperiódica que é atendida pela tarefa servidora. O nível de prioridade $P_1$ se torna ativo e é atribuido à $RT_1$ o valor 6. Em $t = 2.33$, a tarefa aperiódica termina de executar, esgotando a capacidade de serviço da tarefa servidora, e $P_1$ fica inativa. O retorno de 1.33 unidades de tempo será fei-

to no instante 6. Em t = 3.33, $T_1$ completa seu processamento e $T_2$ começa a executar. Em t = 6, ocorre a primeira devolução de tempo consumido, voltando a capacidade total da tarefa servidora para 1.33 unidades de tempo. Em t = 8, ocorre a segunda invocação de tarefa aperiódica, $P_1$ se torna ativa e a tarefa servidora atende a invocação. $RT_1$ recebe o valor 13. Em t = 9.33, após a tarefa aperiódica ter sido executada, $P_1$ fica inativa, e $T_2$ retoma seu processamento. Em t = 13, ocorre a segunda devolução de tempo de execução para a tarefa servidora, trazendo novamente sua capacidade de serviço a 1.33 unidades.

Tarefas aperiódicas com "hard deadline" não são geralmente suportadas pelos algoritmos vistos anteriormente. Para garantir o cumprimento do prazo destas tarefas, podem ser criadas tarefas servidoras de alta prioridade, uma para cada tarefa aperiódica. Para certificar-se que a tarefa servidora sempre tenha tempo de execução suficiente para o cumprimento do prazo da tarefa aperiódica, deve-se impor uma restrição ao tempo mínimo entre invocações desta tarefa. O tempo mínimo entre invocações deve ser igual ou maior que o período da tarefa servidora. Neste caso, o algoritmo SS pode ser utilizado para garantir o cumprimento do prazo das tarefas aperiódicas, contanto que este prazo seja maior ou igual ao tempo mínimo entre invocações. Entretanto, se o prazo for menor que o tempo mínimo entre invocações, é necessário uma atribuição de prioridades diferente para a tarefa servidora. A prioridade da tarefa servidora deve ser baseada no prazo da tarefa aperiódica, a qual está associada, e não no tempo mínimo entre invocações dela. Este tipo de atribuição de prioridade é conhecido como *prazo monotônico* [LEU 82], e é empregada no algoritmo "deadline monotonic sporadic server".

#### 2.4.2.6 - "Deadline Monotonic Sporadic Server" (DMSS)

Com exceção da atribuição de prioridade da tarefa servidora, a operação do algoritmo DMSS é identica ao algoritmo SS. A tarefa servidora é criada com a capacidade de serviço igual ao tempo de execução da tarefa aperiódica, a qual deve tratar. A prioridade da tarefa servidora é baseada no prazo da tarefa que ela está servindo. Assim, o algoritmo *taxa monotônica* é utilizado para atribuir prioridades às tarefas periódicas, e a prioridade da tarefa servidora é atribuida como se seu período fosse igual ao prazo da tarefa aperiódica associada.

O exemplo dado a seguir ilustra o esquema de atribuição de prioridades do algoritmo DMSS. As tarefas são dadas pelas tabelas 2.12 e 2.13. Foi criada uma tarefa servidora com tempo de execução igual a 8 unidades e um período de 32 unidades (igual ao tempo mínimo entre invocações da tarefa aperiódica associada). A tarefa servidora possui a maior prioridade, pois é considerado, quando feita a atribuição de prioridades, que o período dela é 10 unidades ("deadline" da tarefa aperiódica tratada) e não 32 (seu período real).

| Tarefas Periódicas | Tempo de Execução | Período | Deadline |
|---|---|---|---|
| $T_1$ | 4 | 12 | 12 |
| $T_2$ | 4 | 20 | 20 |
| S | 8 | 32 | 10 |

Tabela 2.12 - Especificação das tarefas periódicas

| Tarefas Periódicas | Tempo de Execução | MEI | Deadline |
|---|---|---|---|
| $T_3$ | 8 | 32 | 10 |

Tabela 2.13 - Especificação da tarefa aperiódicas

MEI = mínimo entre invocações

A figura 2.8 mostra o diagrama de execução das tarefas segundo o algoritmo DMSS. Nota-se que a tarefa aperiódica consegue cumprir seu prazo. A figura 2.9 ilustra a execução do mesmo conjunto de tarefas para o algoritmo *taxa monotônica*. Pelo esquema de atribuição de prioridades deste algoritmo, a tarefa servidora tem a menor prioridade, o que leva à violação de prazo no instante 10. Este exemplo mostra a importância do algoritmo DMSS para garantir as tarefas aperiódicas, cujos prazos são menores que os respectivos tempos mínimos entre invocações.

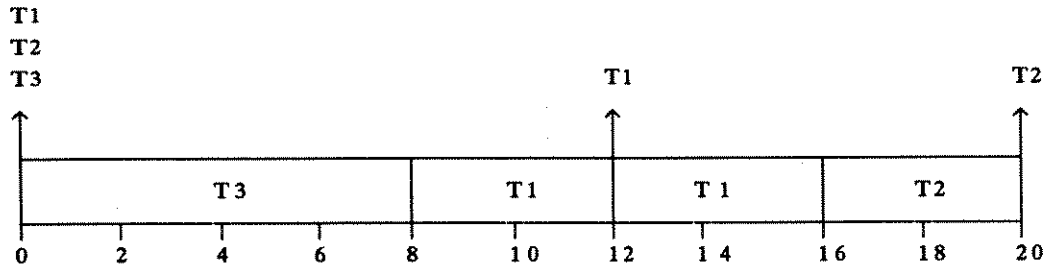



Figura 2.8 - Execução do algoritmo DMSS

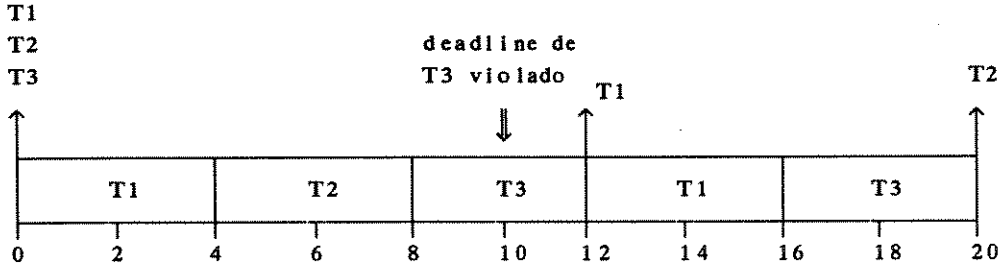



Figura 2.9 - Execução do algoritmo *taxa monotônica* com tarefa aperiódica com deadline menor que o tempo mínimo entre invocações

Um conjunto de equações para análise de escalonamento, desenvolvido para o protocolo "topo de prioridades" [SHA 87] que trabalha em conjunto ao algoritmo *taxa monotônica*, pode ser usado para testar a escalonabilidade de tarefas aperiódicas com "hard deadline" e atribuição de prioridades dada pelo algoritmo DMSS.

Os algoritmos apresentados até o momento são do tipo estático, isto é, as prioridades são determinadas antes de iniciada a execução e são fixas. A seguir são vistos dois algoritmos dinâmicos.

### 2.4.3 - "Earliest Deadline"

O algoritmo *"earliest deadline"* introduzido por Liu e Layland [LIU 73] possui esquema dinâmico de atribuição de prioridade às tarefas (prioridades não-fixas). As tarefas que o algoritmo considera são periódicas , independentes (sem restrições de precedência ou comunicação), preemptíveis e possuem prazo ("deadline") para terminar de executar.

No algoritmo *"earliest deadline"*, a prioridade das tarefas é baseada no respectivo prazo ("deadline") que elas têm para cumprir. A tarefa com o menor prazo para terminar de executar, ou seja, com o "deadline" mais próximo a vencer, possui a maior prioridade entre as tarefas do sistema e vice-versa. A atribuição de prioridades é feita a cada chegada de tarefa.

O "deadline" de uma tarefa periódica necessariamente tem o valor igual ou menor do que o seu período. Quando é dado um "deadline" para a tarefa, este valor é relativo, isto é, se uma tarefa possui um "deadline" D, a partir do instante em que for invocada ela terá D unidades de tempo para terminar de executar. O "deadline" absoluto é a soma do instante de invocação da tarefa com o valor do "deadline" relativo.

Como exemplo, considere duas tarefas $T_1$ e $T_2$ com seus respectivos parâmetros dados na tabela 2.14.

O diagrama de Gantt na figura 2.10 mostra a execução das tarefas entre os instantes 0 e 60.

| Tarefas Periódicas | Execução | Período | Deadline |
|---|---|---|---|
| $T_1$ | 10 | 20 | 20 |
| $T_2$ | 25 | 50 | 50 |

Tabela 2.14 - Tarefas Periódicas

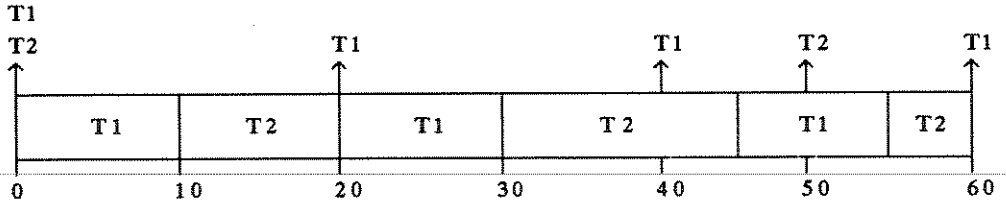



Figura 2.10 - Execução do algoritmo *"earliest deadline"*

Inicialmente T1 e T2 estão prontas para executar; como o "deadline" de T1 (20) é menor do que o de T2 (50), T1 obtém maior prioridade, executando até o instante 10 quando então termina e cede o processador à T2. No instante 20, T2 é interrompida por uma nova invocação de T1 que novamente recebe prioridade máxima e executa até 30 quando então retorna T2 para terminar de processar. Em 40, chega uma nova instância de T1 mas, como seu "deadline" (60) é maior que o de T2 (50), T1 não ganha a CPU e T2 consegue finalizar seu processamento. Deve ser observado que a decisão da atribuição é sempre realizada quando chega uma tarefa, pois é o único instante possível de ocorrer alteração de prioridades entre as tarefas do sistema.

Dado um conjunto de n tarefas periódicas, pode-se verificar se, empregando o algoritmo *"earliest deadline"*, a execução delas estará dentro dos respectivos prazos. Para isto é suficiente checar a seguinte relação :

$$C_1/P_1 + \ldots + C_n/P_n \leq 1 \quad \text{[LIU 73]}$$

onde

$C_i$ = tempo de execução da tarefa i

$P_i$ = período da tarefa i

Este resultado mostra que o limite do fator de utilização da CPU para o algoritmo *"earliest deadline"* é de 100%, o que representa uma boa melhora em relação ao valor apresentado para o algoritmo *taxa monotônica*.

### 2.4.4 - "Least Laxity"

O algoritmo *"least laxity"* possui esquema de atribuição dinâmico de prioridades às tarefas. Este algoritmo trabalha com tarefas periódicas , independentes (sem restrição de precedência ou comunicação), preemptíveis e que possuem "deadline" para cumprir.

O algoritmo *"least laxity"* baseia-se no conceito de folga ("laxity") das tarefas. Supondo que uma tarefa comece a sua execução no instante atual, sua folga é definida como o intervalo compreendido entre o instante em que terminará de executar e o seu "deadline", considerando que não seja interrompida. A folga de uma tarefa T

num instante t pode ser calculada pela fórmula:

$$F_T(t) = D - (t+CR)$$

onde

$F_T(t)$ = folga da tarefa T no instante t;
D = "deadline" absoluto da tarefa T
t = instante atual;
CR = tempo de execução restante p/ a tarefa terminar de processar.

A folga representa o tempo máximo que pode ser adiada a execução da tarefa, de forma que ainda seja respeitado seus requisitos temporais.

Por exemplo, suponha que o instante atual seja 10 unidades de tempo, uma tarefa T tenha "deadline" em 20, e restam a ela 4 unidades de tempo para completar seu processamento. A folga da tarefa T no instante 10 será:

$$F_T(10) = 20 - (10 + 4) = 6$$

No algoritmo *"least laxity"*, a prioridade maior é dada à tarefa com menor folga e vice-versa. Para ilustrar o seu funcionamento, considere o exemplo a seguir. A figura 2.11 mostra a execução de um conjunto de tarefas, cujas características é dada pela tabela 2.15. A tabela 2.16 apresenta a variação da folga das tarefas nos instantes de execução. Deve ser observado que a folga da tarefa que se encontra executando mantem-se constante, enquanto que a folga das que aguardam a vez de executar decresce.

| Tarefas Periódicas | Execução | Período | Deadline |
|---|---|---|---|
| $T_1$ | 4 | 7 | 7 |
| $T_2$ | 2 | 6 | 6 |

Tabela 2.15 - Especificação das tarefas periódicas

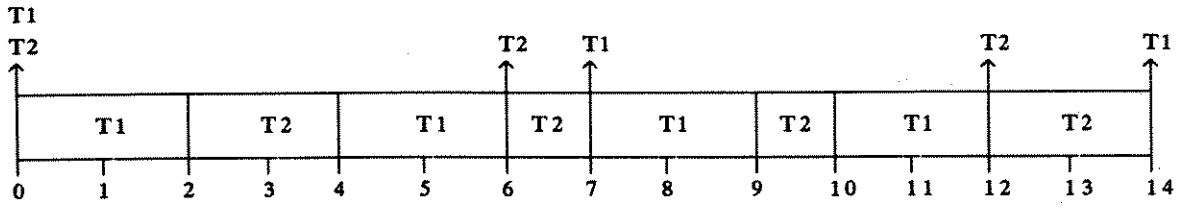



Figura 2.11 - Execução do algoritmo *"least laxity"*

| Tempo | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Folga | | | | | | | | | | | | | | |
| T1 | 3 | 3 | 3 | 2 | 1 | 1 | - | 3 | 3 | 3 | 2 | 2 | - | - |
| T2 | 4 | 3 | 2 | 2 | - | - | 4 | 4 | 3 | 2 | - | - | 4 | 4 |

Tabela 2.16 - Folga das tarefas

Como o evento que determina a mudança de prioridade das tarefas (uma tarefa ter menor folga do que aquela que está executando) pode ocorrer a qualquer instante, a política "least laxity first" necessita de monitoramento a cada unidade de tempo, o que pode representar um ponto negativo ao algoritmo do ponto de vista da sua implementação, apesar de ser considerado ótimo.

## 2.5 - Conclusão

Este capítulo apresentou alguns algoritmos de escalonamento importantes encontrados na literatura. Porém, as suposições feitas por estes algoritmos em relação às tarefas (independentes, tempo de pronto = tempo de chegada, etc.), restringem bastante as aplicações reais onde podem ser utilizados. É comum, nos sistemas de tempo real (crítico e não-crítico), as tarefas necessitarem de troca de dados, sincronismo, compartilhamento de recursos, terem relações de precedência entre si, etc. Tais requisitos aumentam a complexidade do problema de escalonamento, levando às vezes a um problema NP-hard.

Quando não é possível encontrar regras formais, que provam precisamente a escalonabilidade de um conjunto de tarefas, os algoritmos de escalonamento empregam técnicas heurísticas. Devido a sua natureza, tais algoritmos precisam testar a execução da aplicação várias vezes até chegar a um resultado. Neste sentido, alguns pontos devem ser considerados:

- existe disponível um núcleo de tempo real com a política de escalonamento compatível com as características das tarefas da aplicação?
- supondo que exista, o núcleo oferece facilidades de análise que permitam, por exemplo, detectar violação de prazos das tarefas?
- caso não exista, deve-se optar entre a construção de um núcleo de tempo real com as especificações exigidas ou, a adaptação de um núcleo que se tenha disponível. Esta última opção geralmente é complicada, podendo requerer um novo projeto do núcleo;
- dependendo da aplicação, o teste de sua execução pode levar um longo tempo.

Os aspectos citados permitem concluir que, testar uma aplicação executando-a diretamente sob um núcleo de tempo real é pouco viável. Uma alternativa é uma ferramenta de software que simule a execução de tarefas especificadas pelo usuário, tenha disponível várias políticas de escalonamento e possua facilidades de análise de escalonamento.

Devido à importância das estratégias de escalonamento para os sistemas de tempo real crítico, novos algoritmos têm sido propostos na literatura. Para se conhecer melhor a eficiência destes algoritmos, são realizados muitos testes. Um ambiente que simplificasse a implementação de escalonadores (dispensando o conhecimento completo dos detalhes de um núcleo de tempo real), e permitisse posteriormente estudá-los de uma forma interativa, seria de grande utilidade.

O próximo capítulo apresentará uma ferramenta de software proposta para atender as reinvidicações colocadas nesta seção.

# Capítulo III

# Sistema de Simulação para Escalonadores

# CAPÍTULO III

# SISTEMA DE SIMULAÇÃO PARA ESCALONADORES

## 3.1 - Introdução

O desenvolvimento de sistemas de tempo real cada vez mais complexos tem inspirado a busca de ferramentas de software adequadas que auxiliem esta tarefa.

Em particular, os chamados sistemas de tempo real crítico ("hard real time systems") demandam por ferramentas que levem em consideração os rígidos requisitos de tempo impostos para o funcionamento correto da aplicação. Apesar de serem ainda poucas, já existem algumas ferramentas automatizadas que facilitam, sob algum aspecto, o desenvolvimento destes sistemas: SARTOR [MOK 85], Scheduler 1-2-3 e ARM [TOK 88], Projeto COINS 520 [GEN 89].

O SARTOR tem por objetivo fornecer um ambiente que possibilite o desenvolvimento correto de software de tempo real. O SARTOR incorpora uma linguagem de especificação (Modechart [STU 89]) dedicada a construção de sistemas de tempo real. Também possui uma ferramenta para a verificação (Verify4 [STU 89]) de certas propriedades, presentes na especificação do sistema usando a linguagem Modechart.

O Scheduler 1-2-3 e o ARM formam um conjunto integrado de ferramentas para o núcleo de tempo real ARTS [TOK 88], construido para uma rede de "Workstations SUN3". O Scheduler 1-2-3 verifica se um conjunto de tarefas com "hard deadlines" podem executar, obedecendo determinada política de escalonamento, cumprindo seus respectivos prazos. O ARM (Advanced Real-Time Monitor) permite visualizar o comportamento de um sistema de tempo real durante sua execução, levando em consideração "overheads" introduzidos pelo monitoramento do sistema. Ambas ferramentas são utilizadas na fase de projeto do sistema.

Construido para o projeto SPRING [RAM 89], o projeto COINS 520 consiste de uma ferramenta que simula um sistema operacional de tempo real em um ambiente distribuído e permite avaliar algoritmos de escalonamento, configurações de hardware e cargas do sistema.

Nas aplicações de tempo real crítico, um ponto muito importante a ser considerado é a política de escalonamento de tarefas implementada no núcleo de tempo real utilizado. É necessário verificar se esta cumpre as necessidades da aplicação. Caso isto não ocorra, deve-se revisar os requisitos de tempo e/ou tentar outra política de escalonamento.

Pela sua importância, muito esforço tem sido dispendido na área de escalonadores para sistemas de tempo real crítico que, como resultado, tem gerado novos algoritmos de escalonamento ou mesmo variações de outros já conhecidos.

Vários destes algoritmos são de natureza heurística, e assim, para se conhecer melhor suas características e também utilidades, é preciso testá-los exaustivamente.

Os aspectos citados motivaram a criação de um sistema que simula o funcionamento de um núcleo de tempo real, denominado *Sistema de Simulação para Escalonadores* (*SSE*), cujas principais aplicações são:

1 - Ferramenta de análise de escalonamento de tarefas;
2 - Ambiente para implementação de políticas de escalonamento.

A primeira opção de uso para o SSE possibilita que projetistas de software, conhecendo as características do sistema a ser implementado, analisem o comportamento das tarefas que o compõem, em relação ao escalonamento, verificando se os requisitos de tempo serão satisfeitos. Para isto, o SSE dispõe de uma série de facilidades que serão discutidas na próxima seção.

O SSE possui um conjunto de escalonadores, dentre os quais o usuário pode escolher o mais apropriado para utilizar. Entretanto, o sistema não se restringe a apenas estes escalonadores. Em função da arquitetura empregada em sua construção, o SSE também serve de ambiente para testar implementações de políticas de escalonamento escritas pelo próprio usuário, através da utilização de uma interface de software bem definida. Em relação a este aspecto, o SSE tem duas aplicações principais. A primeira, na área de ensino, permite que a ferramenta seja usada em cursos, com o objetivo de exercitar a implementação de algoritmos de escalonamento e estudar seu funcionamento. A outra aplicação, na área de pesquisa, demonstra que o SSE pode ter um uso promissor no estudo de novos algoritmos de escalonamento.

Apesar da versatilidade que o SSE apresenta para trabalhar com outros escalonadores, além dos já incorporados, foi necessário limitá-los a uma classe, segundo algumas características, devido à grande variedade de algoritmos de escalonamento existente. O capítulo II mostrou uma classificação destes algoritmos, conforme o ambiente em que operam e também o tipo de tarefas com que trabalham. Neste contexto, o SSE é dedicado a escalonadores que executam sob ambiente centralizado com monoprocessador. As tarefas que participam da simulação podem ser de natureza periódica ou aperiódica, terem prazo ou não para cumprir ("hard" e "soft deadline") e, ainda, possuirem comunicação entre si ou serem independentes.

Este capítulo está dividido em duas partes, nas quais são apresentadas as formas de utilização do SSE. Na primeira parte, são mostradas as facilidades e serviços que o sistema dispõe como ferramenta de análise de escalonamento. A seguir, é explicado como o SSE serve de ambiente para implementação de políticas de escalonamento. Detalhes de implementação do SSE serão tratados quando o contexto exigir, sendo a descrição completa deixada para o capítulo IV.

## 3.2 - Ferramenta de Análise de Escalonamento

O SSE é uma ferramenta para ser empregada na fase de testes no desenvolvimento de sistemas de tempo real crítico. Ela simula a execução de um conjunto de tarefas, exibindo a sequência em que são escalonadas, acompanhada de outras informações úteis para a análise de escalonamento. Durante a simulação, o SSE verifica os requisitos de tempo de cada tarefa participante, e acusa qualquer violação de prazo ("deadline") que uma tarefa venha a sofrer. Assim, o usuário, conhecendo as características das tarefas e do sistema em questão, consegue com esta ferramenta fazer a validação dos requisitos temporais da sua aplicação.

A operação do SSE é facilitada por uma série de recursos que tornam o seu uso bastante amigável. Os serviços nele presentes foram agrupados em cinco classes:

1 - Edição de Tarefas
2 - Configuração do Sistema
3 - Execução da Simulação

4 - Análise da Simulação

5 - Serviços de Arquivo

Nas seções seguintes serão descritos estes serviços, procurando ao mesmo tempo mostrar o funcionamento do SSE como ferramenta de análise de escalonamento.

### 3.2.1 - Edição de Tarefas

As tarefas que fazem parte da simulação não são reais, isto é, não possuem um segmento de código a ser executado. Elas são representadas por uma estrutura de dados, onde seus campos contêm informações que caracterizam a tarefa e seu estado dentro da simulação, e que são utilizados pelo SSE. Esta estrutura será apresentada em detalhe na seção 3.3.3.

Existem três operações básicas para manipular as tarefas:

1 - *Inserir*

2 - *Alterar*

3 - *Retirar*

A entrada de novas tarefas no sistema é feita pela operação *Inserir*, onde são informados o tipo de tarefa quanto à natureza (periódica ou aperiódica) e seus atributos. Considera-se atributos de uma tarefas os seguintes parâmetros:

1 - *Identificador*: é um número diferente para a identificação de cada tarefa;

2 - *Tempo de Execução*: tempo máximo que a tarefa pode levar para completar sua execução. (Pior caso de execução);

3 - *Deadline*: prazo relativo que a tarefa tem para cumprir sua execução a partir do instante em que é invocada. O valor 0 indica que a tarefa não possui prazo para cumprir ("soft deadline");

4 - *Período*: intervalo de tempo que repete a invocação de uma tarefa periódica;

5 - *Início*: instante em que será invocada uma tarefa (este parâmetro é disponível apenas para tarefas aperiódicas);

6 - *Rendezvous*: Se a tarefa possui rendezvous, para cada um especificado, o sistema pede que seja informado:

6.1 - *Destino*: identificador da tarefa com o qual será feito o rendezvous;

6.2 - *Instante:* nº de "ticks" (unidade lógica de tempo) que a executa, relativamente ao seu início até atingir o rendezvous;

Por exemplo, para as tarefas periódicas 1, 2 e 3, com os respectivos tempo de execução 50, 23 e 27 "ticks" e com os rendezvous conforme esquematizados na figura 3.1, os parâmetros de rendezvous têm os seguintes valores:

| TAREFA [1] | TAREFA [2] | TAREFA [3] |
|---|---|---|
| Rendezvous[1] | Rendezvous[1] | Rendezvous[1] |
| Destino = 2 | Destino = 1 | Destino = 1 |
| Instante = 15 | Instante = 7 | Instante = 18 |
| Rendezvous[2] | | |
| Destino = 3 | | |
| Instante = 35 | | |

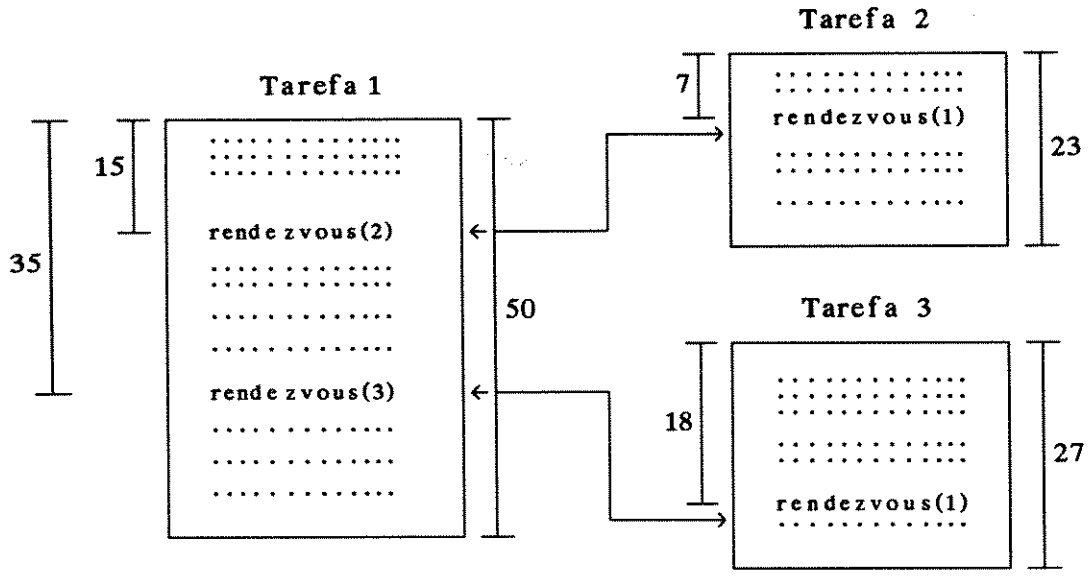



Figura 3.1 - Tarefas com rendezvous

7 - *Comandos Genéricos*: A possibilidade de se especificar *comandos genéricos* permite simular a ocorrência de comandos que resultam em um escalonamento de tarefas. O tratamento deste comando é de inteira responsabilidade do módulo escalonador, sendo que a única operação que o SSE executa é a sinalização indicando que aconteceu tal evento. Um exemplo de aplicação de *comando genérico* seria para simular as operações de semáforo. Será discutido na seção 3.3 o uso dos *comandos genéricos*.

Para cada *comando genérico* especificado, é necessário informar:

7.1 - *Tipo de Comando*: é um número que identifica o *comando genérico*. Os valores válidos são de 1 a 64000;

7.2 - *Instante*: nº de "ticks" que a tarefa executa até atingir este comando (análogo ao mesmo parâmetro usado para definir instante de rendezvous);

7.3 - *Parâmetros:* parâmetros pertencentes ao comando. Para cada parâmetro (se houver) deve-se informar um valor numérico inteiro.

As estruturas de dados associadas aos *comandos genéricos* são alocadas dinamicamente, assim, a quantidade de comandos que uma tarefa suporta é limitada apenas pelo espaço de memória disponível. O mesmo é válido para os parâmetros dos *comandos genéricos*, quando houverem. Quando se estiver utilizando um dos escalonadores internos do SSE, o preenchimento destes campos não causará ação alguma, pois tais algoritmos de escalonamento não trabalham com *comandos genéricos*.

8 - *Prioridade*: é um valor inteiro que indica a prioridade da tarefa. O preenchimento deste campo é opcional, pois depende se a implementação da política de escalonamento o utiliza ou não. Entre os algoritmos de escalonamento internos do SSE, o de *Prioridade Fixa Simples* é o único que faz uso deste parâmetro.

9 - *Campos Auxiliares*: pode acontecer que algum algoritmo de escalonamento implementado por um usuário leve em consideração parâmetro(s) diferente(s) dos que as tarefas já possuem. Para isto são oferecidos Campos Auxiliares, os quais podem receber valores numéricos que serão interpretados pelo escalonador. Existem dois conjuntos de Campos Auxiliares que diferem pelo tamanho do campo:

*Campos Auxiliares I*: tamanho 1 byte (tipo *unsigned char*);

*Campos Auxiliares II*: tamanho 4 bytes (tipo *unsigned long*).

Estes parâmetros não são utilizados pelos escalonadores internos do SSE.

Assume-se que todos os parâmetros das tarefas relacionados com tempo (a saber, *tempo de execução*, *deadline*, *período*, *início*, *instante de rendezvous* e *instante de comando genérico*) são dados em nº de "ticks". O termo "tick" é aqui empregado para expressar unidade de tempo lógico. A escolha do valor adequado para esta unidade, em relação a uma base de tempo, é de responsabilidade do usuário. Por exemplo, um "tick" pode ser equivalente a 30 ms ou a 1 min. A unidade dependerá da ordem de grandeza do tempo de execução das tarefas com as quais se está trabalhando.

Após ter sido inserida uma tarefa, é possível modificar qualquer um de seus parâmetros, utilizando a operação *Alterar*. Os campos da estrutura de dados da tarefa são atualizados com o(s) valor(es) modificado(s).

Quando é desejado, por algum motivo, que determinada tarefa não faça mais parte da simulação, a tarefa especificada pode ser removida do sistema através da operação *Retirar*.

As três operações descritas acima podem ser usadas antes do início ou durante a simulação. Este aspecto é interessante pois permite, por exemplo, inserir novas tarefas com o sistema "executando".

Antes de começar uma simulação, além de inseridas as tarefas, é preciso passar algumas informações ao SSE para configurar o sistema.

### 3.2.2 - Configuração do Sistema

Existem alguns parâmetros do SSE a serem especificados, que estão relacionados diretamente com a simulação:

1 - *Tempo de Simulação*
2 - *Modo de Execução*
3 - *Política de Escalonamento*
4 - *Breakpoint*

O *Tempo de Simulação* refere-se até que instante a simulação deverá ocorrer. Este valor é dado em nº de "ticks", como no caso de alguns dos parâmetros das tarefas. Por exemplo, para o *Tempo de Simulação* = 5000 "ticks", será mostrada a sequência de escalonamento entre o instante 0 e aquele valor.

O *Modo de Execução* especifica o tipo de interação com o usuário durante a execução da simulação. Existem dois modos: *interativo* e *não-interativo*. No primeiro, a cada evento significativo ao SSE, é feita uma interrupção da execução que permite ao usuário acessar as facilidades do sistema. O conceito de evento para o SSE será visto quando da descrição da opção *Execução da Simulação*. O modo *interativo* possibilita, por exemplo, inserir uma tarefa (periódica ou aperiódica), mudar a configuração ou acessar qualquer dos demais serviços disponíveis durante a simulação.

Já no modo *não-interativo*, a execução transcorre sem intervenção por parte do usuário. Ao contrário do modo *interativo*, a simulação não é interrompida a cada evento que acontece. Este modo é adequado quando não se deseja qualquer tipo de inte-

ração com o SSE, isto é, não são utilizados os serviços disponíveis do sistema durante a simulação.

No SSE existem algumas políticas de escalonamento já implementadas. Isto permite analisar o comportamento de um conjunto de tarefas, utilizando-se diferentes algoritmos de escalonamento, e decidir qual o mais adequado aos requisitos da aplicação. Atualmente, seis políticas estão implementadas: "Earliest Deadline", "Least Laxity", Taxa Monotônica, "Deferrable Server", FIFO, Prioridade Fixa Simples. Além destas opções de escalonadores é possível introduzir um novo escalonador fornecido pelo usuário, caso este queira executar alguma política diferente das disponíveis. Esta particularidade permite, através de uma interface de software bem definida, que sejam analisados outros algoritmos de escalonamento, tornando o SSE uma ferramenta flexível. O procedimento para se trabalhar com esta opção merece um detalhamento à parte, o que será feito na seção 3.3.

Um recurso bastante útil, quando se está depurando um programa, são os chamados "breakpoints", que são pontos especificados no software que, quando atingidos, fazem com que a execução seja suspensa temporariamente, permitindo ao usuário tomar alguma ação (por exemplo, ver o valor de uma variável). Em analogia, foi criado um "*Breakpoint*" de simulação no SSE, que é definido informando-se o instante de tempo (em "ticks") em que ocorrerá. Quando a execução atinge o "*Breakpoint*", isto é ,o instante escolhido, o SSE suspende a simulação, e aguarda uma intervenção do usuário, que pode ser um simples pedido para continuar ou a utilização de algum dos serviços descritos. O "*Breakpoint*" é um parâmetro opcional a ser informado.

Aos parâmetros *Tempo de Simulação* e "*Breakpoint*" podem ser atribuidos novos valores após iniciada a simulação (contanto que se esteja trabalhando no modo *interativo*).

Com as tarefas já inseridas e a configuração especificada, o SSE está apto a começar a simulação.

### 3.2.3 - Execução da Simulação

O SSE reconhece um conjunto de eventos nos quais se baseia para gerenciar a simulação. Os eventos presentes no sistema são:

1 - Invocação de tarefa (periódica e aperiódica)
2 - Fim de execução de tarefa
3 - Requisição de rendezvous
4 - Resposta à rendezvous
5 - Violação de deadline
6 - Breakpoint
7 - Comando genérico
8 - Evento do usuário
9 - Fim da Simulação

O nome dos eventos dispensa maiores explicações sobre seus respectivos significados, com exceção do "evento do usuário". Este sinaliza uma invocação do módulo escalonador fornecido pelo usuário e o seu significado é particular à implementação da política de escalonamento. Este evento será compreendido melhor, quando for discutido na segunda parte deste capítulo a implementação de algoritmos de escalonamento e sua integração ao SSE.

O resultado da simulação é uma tabela que apresenta a evolução da execução das tarefas, mostrando a sequência de escalonamento.

Na simulação, uma tarefa é "executada" até o instante em que ocorre um dos eventos conhecidos do SSE, quando então é mostrada na tela uma linha representando a execução da tarefa, que contém as seguintes informações:

1 - Tempo atual
2 - Tipo de tarefa (periódica ou aperiódica)
3 - Identificador da tarefa
4 - Até que instante a tarefa executou
5 - Deadline absoluto
6 - Tempo restante necessário p/ a tarefa terminar de executar
7 - Evento ocorrido

Este tipo de resultado possibilita a análise do comportamento das tarefas segundo a política de escalonamento utilizada, sendo equivalente a um diagrama de Gantt.

Como exemplo, considere o conjunto de tarefas da tabela 3.1 com seus respectivos parâmetros:

| Tarefas Periódicas | | |
|---|---|---|
| Ident. | 1 | 2 |
| Tempo Exec. | 10 | 25 |
| Deadline | 20 | 50 |
| Período | 20 | 50 |
| Rendezvous | -- | -- |
| Com. Genér. | -- | -- |
| Prioridade | -- | -- |
| Campos Aux. | -- | -- |

| Tarefa Aperiódica | |
|---|---|
| Ident. | 3 |
| Tempo Exec. | 5 |
| Deadline | 10 |
| Início | 35 |
| Rendezvous | -- |
| Com. Genér. | -- |
| Prioridade | -- |
| Campos Aux. | -- |

Tabela 3.1 - Especificação de tarefas

O escalonamento representado pelo diagrama de Gantt e o resultado correspondente gerado pelo SSE são vistos nas figuras 3.2 e 3.3, respectivamente. A política empregada é a "Earliest Deadline".

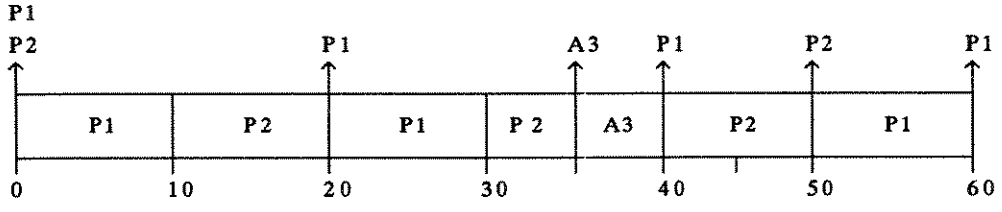



Figura 3.2 - Diagrama de Gantt

| Tempo | Tarefa | Executa até | Deadline | Exec.Rest. | Evento |
|---|---|---|---|---|---|
| 0 | P1 | 10 | 20 | 0 | fim exec. |
| 10 | P2 | 20 | 50 | 15 | invoc. tar. |
| 20 | P1 | 30 | 40 | 0 | fim exec. |
| 30 | P2 | 35 | 50 | 10 | invoc. tar. |
| 35 | A3 | 40 | 45 | 0 | fim exec. |
| 40 | P2 | 50 | 50 | 0 | fim exec. |
| 50 | P1 | 60 | 60 | 0 | fim simul. |

Figura 3.3 - Resultado da simulação gerado pelo SSE

Na tabela acima, a letra "P" ou "A" que antecede o identificador da tarefa, indica que ela é de natureza periódica ou aperiódica, respectivamente.

A interpretação do resultado gerado pelo SSE seria:

- linha 1: a tarefa periódica 1 executa do instante 0 ao 10; seu "deadline" é em 20; o tempo restante de execução é 0 (isto é, terminou sua execução) e o evento significativo ao SSE foi o fim de execução da tarefa.
- linha 2: a tarefa periódica 2 executa do instante 10 ao 20; seu "deadline" é em 50; restam 15 "ticks" para terminar sua execução e o evento que causou a preempção foi a chegada de uma tarefa. Desta forma segue a análise até o fim da simulação.

Se na configuração o modo de execução for *interativo*, cada vez que ocorre um evento conhecido do SSE, a simulação é interrompida permitindo o acesso a qualquer das facilidades do sistema. Caso isto aconteça, é possível reiniciar a simulação ou utilizar a opção "*Continue*" que fará com que a execução retorne a partir do ponto em que havia parado.

A ocorrência de violação do "deadline" de alguma tarefa, causa uma sinalização e a simulação é interrompida e finalizada. Esta decisão parece coerente visto que tal evento, em geral, é de significativa importância para ser ignorado e deixar que a simulação continue. Tarefas que perdem seu "deadline" são guardadas em uma fila especial que pode ser consultada posteriormente.

Durante e após a simulação, pode-se acessar uma série de serviços que auxiliam a análise do escalonamento das tarefas.

### 3.2.4 - Análise da Simulação

O SSE apresenta algumas facilidades para a análise da simulação. Os serviços oferecidos são:

1 - *Consulta às Filas de Tarefas*
2 - *Verificação de Fórmulas de Escalonamento*
3 - *Utilização de CPU*
4 - *Tempo de Resposta*
5 - *Eventos Ocorridos*

Nas seções seguintes são apresentados cada um destes serviços.

#### 3.2.4.1 - Consulta às Filas de Tarefas

Como em um núcleo de tempo real, o SSE possui filas de tarefas que são empregadas no gerenciamento do sistema. Algumas destas filas são de interesse do usuário, em particular aqueles que estão testando uma política não implementada internamente. Com este objetivo, o SSE dispõe de um serviço de consulta a filas específicas. As filas para consulta são:

1 - Fila de Pronto
2 - Fila de Escalonamento Periódico
3 - Fila de Escalonamento Aperiódico
4 - Fila de Deadlines Violados
5 - Filas Auxiliares

Cada uma destas filas será vista em detalhe quando for discutida as estruturas de dados do SSE . A nível de introdução, seguem seus respectivos significados.

1 - *Fila de Pronto*: contém as tarefas já escalonadas e prontas para executar. A tarefa é dita pronta, se foi invocada, se todos os recursos que necessita estão disponíveis e se não se encontra bloqueada.

2 - *Fila de Escalonamento Periódico*: contém as tarefas periódicas que já foram invocadas e aguardam para serem escalonadas.

3 - *Fila de Escalonamento Aperiódico*: contém as tarefas aperiódicas que já foram invocadas e aguardam para serem escalonadas.

4 - *Fila de Deadlines Violados:* contém as tarefas que tiveram seus prazos violados quando o SSE interrompe por causa deste evento.

5 - *Filas Auxiliares:* são filas disponíveis para utilização pelo usuário, ou seja, possuem significados particulares. Como existe mais de uma fila auxiliar, deve-se especificar a fila desejada.

Os dados apresentados, quando se está consultando uma das filas acima, são os seguintes:

1 - Tarefas que estão na fila (tipo: "P" ou "A" e identificador);
2 - Instante de invocação;
3 - Deadline absoluto;
4 - Tempo restante de execução.

As tarefas apresentadas seguem a mesma ordem em que se encontram na respectiva fila.

### 3.2.4.2 - Verificação de Fórmulas de Escalonamento

Certas políticas de escalonamento apresentam resultados matemáticos que auxiliam na análise de escalonabilidade das tarefas. Alguns destes resultados, representados por fórmulas, foram implementados no SSE. Observa-se que estas fórmulas são particulares de políticas de escalonamento específicas e, assim, só devem ser consultadas quando se estiver utilizando o escalonador correspondente, pois do contrário o valor obtido pode não ter sentido.

As fórmulas implementadas e a respectiva política em que podem ser empregadas são as seguintes:

1 - Fator de Utilização:

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

onde:

n = nº de tarefas periódicas

$C_i$ = tempo de execução da tarefa i

$T_i$ = período da tarefa i;

Algoritmos: Taxa Monotônica, "Earliest Deadline", "Least Laxity"

2 - Limite de Utilização Garantido:

$$LU(n) = n(2^{1/n} - 1)$$

onde:

n = número de tarefas periódicas

Algoritmo: Taxa Monotônica

Observa-se, como já estudado no capítulo II, que os resultados acima aplicam-se quando se está trabalhando apenas com tarefas periódicas e independentes.

### 3.2.4.3 - Utilização de CPU

A *Utilização de CPU* é uma medida que expressa a taxa de tempo que esta esteve executando algum processamento durante um intervalo.

O SSE fornece três medidas:

1 - Utilização de CPU p/ tarefas periódicas

$$U_p(t) = \frac{SC_p^t}{T_t}$$

2 - Utilização de CPU p/ tarefas aperiódicas

$$U_a(t) = \frac{SC_a^t}{T_t}$$

3- Utilização de CPU total

$$U_t(t) = U_p(t) + U_a(t)$$

onde

$SC_p^t$ = tempo que o processador ficou ocupado com tarefas periódicas até o instante t.

$SC_a^t$ = tempo que o processador ficou ocupado com tarefas aperiódicas até o instante t.

$T_t$ = instante em que está sendo medido a utilização de CPU.

Como exemplo, considere o diagrama de Gantt da figura 3.4:

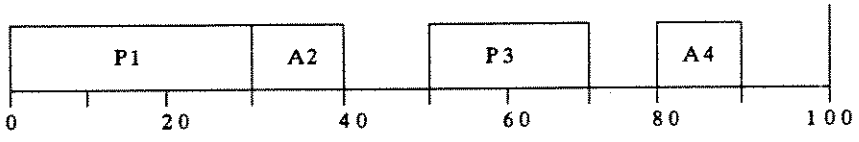



Figura 3.4 - Exemplo - Cálculo de utilização de CPU

Até o instante 100 "ticks" as tarefas periódicas P1 e P3 executaram 30 e 20 "ticks" respectivamente, enquanto que as aperiódicas A2 e A4, 10 "ticks" cada. As medidas de utilização neste caso, considerando o intervalo de 0 a 100 "ticks", serão:

$U_p(t) = (30 + 20) / 100 = 50\%$

$U_a(t) = (10 + 10) / 100 = 20\%$

$U_t(t) = 70\%$

Observa-se que, nos instantes n*mmc(T1,T2,...,Tm), onde n>0 (inteiro) e mmc(T1,T2,...,Tm) é o mínimo múltiplo comum dos períodos das tarefas periódicas do sistema:

$$U_P(t) = U(n) = \sum_{i=1}^{m} (Ci/Ti)$$

### 3.2.4.4 - Tempo de Resposta

O tempo de resposta de uma tarefa é o intervalo de tempo entre o instante em que foi invocada e o término de sua execução. Esta medida é aplicada principalmente nas tarefas aperiódicas, como foi visto quando descritos os algoritmos de Taxa Monotônica que tratavam deste tipo de tarefas.

Quando é consultado o tempo de resposta de uma tarefa periódica, o valor informado é o da última instância executada daquela tarefa.

### 3.2.4.5 - Eventos Ocorridos

Durante a execução de uma simulação, é possível que em um dado instante coincida a ocorrência de mais de um evento. Esta situação é sinalizada no campo "Evento Ocorrido" ,através da mensagem "+ de 1". Para saber quais os eventos que ocorreram neste instante, basta interromper a simulação e selecionar a opção *Eventos*. Esta opção lista o(s) evento(s) que causaram o último escalonamento.

## 3.2.5 - Serviços de Arquivo

O SSE dispõe de cinco serviços relacionados à operação de arquivos:

1 - *Gravar*

2 - *Carregar*

3 - *Relatório*

4 - *Diretório*

5 - *Muda_Dir*

Os parâmetros das tarefas e a configuração do SSE podem ser gravados em um arquivo permitindo que mais tarde sejam carregados para uma nova simulação. Para isto, são utilizadas as opções *Gravar* e *Carregar*, respectivamente.

A opção *Relatório* é bem interessante, principalmente para os usuários que desejam analisar posteriormente o resultado da simulação. Esta opção especifica um arquivo do tipo ASCII, onde será gravada a execução de cada simulação conforme transcorrida. Os dados gravados são os mesmos apresentados na seção 3.2.3.

Como complemento aos serviços de arquivo apresentados acima, as duas últimas opções *Diretório* e *Muda_Dir* permitem consultar e mudar de diretório de trabalho, respectivamente.

Nas seções anteriores, deu-se enfoque ao SSE como ferramenta de análise de escalonamento, onde foram apresentadas as várias facilidades que o sistema comporta. A seguir, será apresentado o suporte que o SSE fornece à implementação de novas políticas de escalonamento.

## 3.3 - Ambiente para Implementação de Políticas de Escalonamento

O SSE é constituido de vários módulos que foram contruidos, empregando-se os conceitos de baixo acoplamento e alta coesão, conforme discutidos na Engenharia de Software [GAN 83]. Resultante desta arquitetura, obteve-se uma interface bem definida entre a rotina de escalonamento e o resto do sistema. Tal característica trouxe uma outra dimensão para utilização do SSE: ambiente de implementação de políticas de escalonamento. Neste tipo de aplicação para o SSE, trabalha-se a nível de um núcleo de tempo real. O objetivo é possibilitar ao usuário a implementação de algoritmos de escalonamento de tempo real crítico, e depois testá-los, executando a simulação como foi discutido na seção anterior. Para isto, é necessário construir um módulo denominado *Módulo de Escalonamento* que contém a rotina do escalonador e outras rotinas auxiliares, todas escritas pelo usuário. Após compilado este módulo, deve-se executar o "link" dele com os demais módulos que compõem o SSE (os detalhes deste processo são encontrados no apêndice A), obtendo assim o código do SSE executável, com o novo escalonador disponível para a análise.

O Módulo de Escalonamento fornecido pelo usuário deve estar de acordo com as especificações da interface definida para o sistema.

### 3.3.1 - Interface SSE - Módulo de Escalonamento

O escalonador é o módulo responsável por indicar ao processador qual a tarefa a ser executada. A escolha da tarefa é baseada em algum critério , sendo efetuada no instante da ocorrência de um evento significativo. Em termos de implementação, um núcleo de tempo real possui filas de tarefas que são utilizadas no gerenciamento do sistema. Algumas destas filas estão relacionadas diretamente com a função de escalonamento. Analogamente, o SSE apresenta um conjunto de filas de tarefas, onde algumas delas fazem parte da interface entre o SSE e o Módulo de Escalonamento, como é mostrado na figura 3.5.

As filas, as quais o escalonador tem acesso, são as seguintes:

1 - *Fila de Escalonamento Periódico:* contém as tarefas periódicas a serem escalonadas;

2 - *Fila de Escalonamento Aperiódico:* contém as tarefas aperiódicas a serem escalonadas;

3 - *Fila de Pronto:* contém as tarefas prontas para executar;

4 - *Filas Auxiliares:* disponíveis para uso particular.

Quando uma tarefa periódica ou aperiódica recebe uma invocação, esta é colocada na respectiva Fila de Escalonamento. Este evento é sinalizado ao escalonador que, por sua vez, retira as tarefas destas filas e as insere na Fila de Pronto, respeitando um critério de prioridade. A Fila de Pronto é organizada em ordem decrescente de prioridade, isto é, as tarefas de maior prioridade são colocadas à frente das tarefas de menor prioridade. A primeira tarefa desta fila é a que será entregue ao processador para "executar". Quando uma tarefa é processada por completo o SSE retira-a da Fila de Pronto.

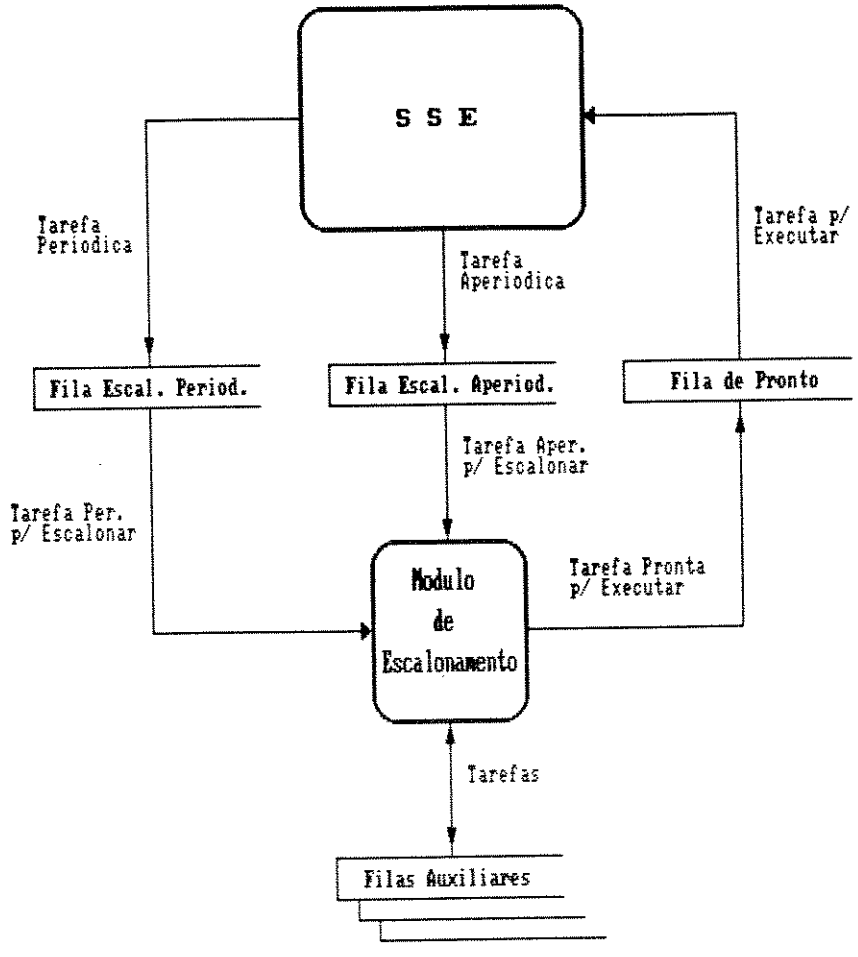



Figura 3.5 - Esquema da interface SSE - Módulo de Escalonamento

No caso de preempção, a tarefa é mantida nesta fila, a menos que tenha feito uma requisição de rendezvous, o que causa a sua retirada e faz com que entre num estado de espera. Quando a tarefa recebe a resposta do rendezvous, esta é colocada na Fila de Escalonamento correspondente, o escalonador é sinalizado e o processo se repete.

Para permitir uma flexibilidade maior, o SSE dispõe de Filas Auxiliares de tarefas que podem ser manipuladas pelo escalonador e cuja finalidade é particular do algoritmo implementado.

As duas filas de escalonamento devem ser atualizadas à medida que as tarefas são escalonadas. Este serviço é de responsabilidade do Módulo de Escalonamento, pois o SSE faz apenas a inserção de tarefas nestas filas. Já com a Fila de Pronto o

procedimento é ao contrário, o escalonador insere tarefas nelas e o SSE trata de sua retirada, quando terminam de executar.

A manipulação destas filas descritas é suportada por um conjunto de funções, que estão disponíveis para a implementação do escalonador e serão apresentadas na seção 3.3.4.

### 3.3.2 - Módulo de Escalonamento

O Módulo de Escalonamento é constituido de três funções implementadas em linguagem C:

1 - *userinit()*
2 - *userinter()*
3 - *scheduler()*

A função *userinit()* é executada uma única vez, no instante de início da execução da simulação. Tem por objetivo inicializar, se necessário, estruturas de dados particulares do Módulo de Escalonamento. Nesta função deve ser indicado se o controle de invocação do escalonador será feito pela função *userinter()* ou pelo SSE. Este controle será explicado na descrição da função *userinter()*. A função *userinit()* também pode invocar alguma outra rotina para, por exemplo, preparar as tarefas para o escalonamento, ou mesmo realizar um escalonamento "off-line". Enfim, qualquer processamento, que necessite ser realizado apenas uma vez e no instante de inicialização da simulação, pode ser feito através desta rotina.

A função *userinter()*, denominada interrupção do usuário, será utilizada, quando assim for especificado na função de inicialização (*userinit()*). Esta rotina é invocada a cada "tick" lógico e retorna um valor booleano que indica se o escalonador (*scheduler()*) será chamado ou não. O objetivo desta função é ter controle sobre eventos não previstos no SSE, que podem causar um escalonamento. Estes eventos desconhecidos são denominados Eventos do Usuário, já citados anteriormente.

São encontradas em núcleos de tempo real rotinas do mesmo tipo cujas funções, em geral, traduzem-se em operar campos temporizados e monitorar o status das tarefas e, quando necessário, invocar o módulo escalonador. Por serem executadas a

cada "tick" do relógio de tempo real, estas rotinas costumam ser bastante simples nos núcleos de tempo real, entretanto, como o SSE trabalha a nível de simulação, a complexidade da função *userinter()* não causa qualquer influência ao desempenho do sistema.

O exemplo a seguir ilustra o uso da função *userinter()*. Suponha que se esteja monitorando um evento, representado por uma variável contendo um valor positivo que é decrementado a cada "tick" lógico. Quando a variável atinge o valor zero, significa que aquele evento ocorreu e determinada tarefa deve ser escalonada. Neste caso, a função *userinter()* tem por objetivo realizar a manutenção da variável que sinaliza a ocorrência do evento. Quando o valor desta variável for zero, a função retorna o valor "true", indicando que o escalonador deve ser invocado, caso contrário é retornado "false". O código C da função *userinter()* para este exemplo é escrito a seguir.

```
#define ANYNUMBER 10000  /* valor que é atribuido à variável  */
                         /* que controla o evento             */
int EventStatus = ANYNUMBER;  /* variável de monitoramento    */
                              /* do evento                    */
Boolean userinter(void)
{
  EventStatus--;          /* atualiza variável                */
  if (EventStatus == 0)   /* Se ocorreu o evento              */
    {
      EventStatus = ANYNUMBER; /* "reset" variável do evento */
      return(true);            /* invoca o escalonador       */
    }
  else                    /* senão                            */
    return(false);        /*     nada feito!                  */
}
```

O algoritmo "Least Laxity" é um exemplo de política de escalonamento cuja implementação utiliza este tipo de gerenciamento. Nele, a prioridade das tarefas é dada pelas respectivas folgas de execução. No decorrer do tempo, este parâmetro varia

para as tarefas prontas que aguardam a vez de executar, enquanto que a que se encontra processando tem a folga constante. Em um dado instante, pode acontecer que a folga de uma tarefa pronta seja menor que a da tarefa que ocupa a CPU, o que deve levar a um reescalonamento. Tal mudança de prioridade não é um evento reconhecido pelo SSE, assim é preciso monitorá-lo de alguma forma. A resposta é a utilização da função *userinter()* que, como é executada a cada "tick" lógico, pode perfeitamente verificar as folgas das tarefas para conferir a prioridade delas e, se necessário, ativar o processo de escalonamento.

Caso seja especificado na função *userinit()* que o controle de invocação do escalonador será feito pelo SSE, a função *userinter()* não terá qualquer efeito pois não será chamada. O SSE invocará o escalonador, apenas se houver tarefa em alguma das Filas de Escalonamento (Periódica e Aperiódica). Esta decisão é tomada a cada ocorrência de um dos eventos que seguem:

1 - Invocação de tarefa (periódica ou aperiódica)
2 - Fim de execução de tarefa
3 - Requisição de rendezvous
4 - Resposta à rendezvous
5 - Breakpoint
6 - Comando genérico

O algoritmo "Earliest Deadline" é um exemplo onde na implementação aplicou-se este tipo de gerenciamento. Nele, a prioridade das tarefas é dada pelos respectivos "deadlines". Os instantes em que pode ocorrer alteração de prioridade são aqueles quando as tarefas sofrem invocação. Como este é um evento que o SSE reconhece, o controle de invocação do escalonador é deixado por sua conta, caso contrário, seria necessário trabalhar com a rotina de interrupção do usuário *userinter()*.

Um exemplo de implementação do Módulo de Escalonamento ilustrando os dois casos de controle de invocação do escalonador, será apresentado ao final deste capítulo.

A última função do módulo, denominada *scheduler()*, é responsável pelo escalonamento das tarefas. Esta rotina é invocada sempre que houver necessidade de atualizar a Fila de Pronto. De um ponto de vista simplista, o procedimento que a função *scheduler()* deve realizar é: retirar as tarefas disponíveis nas Filas de Escalonamen-

to (Periódico e Aperiódico) e, considerando um critério de prioridade definido pela política de escalonamento, inserí-las na Fila de Pronto mantendo-a ordenada. Um exemplo de política, cuja implementação segue este algoritmo simples, é o escalonador "Earliest Deadline" (exemplo I no final do capítulo).

Os algoritmos de escalonamento não possuem necessariamente apenas os passos descritos acima. Eles podem incluir operações mais elaboradas, envolvendo filas de tarefas auxiliares ou outras estruturas de dados, assim como chamadas a funções dedicadas ao algoritmo.

As três funções que compõem o Módulo de Escalonamento trabalham as tarefas através da manipulação de seus campos de dados. A seguir será descrita a estrutura de uma tarefa.

### 3.3.3 - Estrutura da Tarefa

Cada tarefa no SSE é representada por uma estrutura de dados denominada Bloco de Controle da Tarefa (BCT). O BCT é formado de vários campos relacionados com as características e o estado de uma tarefa. Alguns destes campos são de interesse para a interface SSE-Módulo de Escalonamento. Abaixo encontra-se a estrutura de dados do BCT implementada no SSE na linguagem C .

```
typedef struct BCT{
                    unsigned char TaskId;
                    enum PerT     PerType;
                    enum TSt      Status;
                    unsigned long RDeadline;
                    unsigned long Deadline;
                    unsigned long Period;
                    TaskPtr       SchNext;
                    unsigned long ExecTime;
                    unsigned long RemExec;
                    TaskPtr       ReadyNext;
                    unsigned long Start;
                    unsigned long Finish;
```

```
    unsigned char Priority;
    TaskPtr       AuxNext[MAXQ];
    unsigned char Aux1[MAXVARS];
    unsigned long Aux2[MAXVARS];
    ComPtr        Command;
};
```

O BCT apresentado contém apenas os campos vistos pela interface. Segue a descrição de cada campo:

*TaskId*: identificador da tarefa; único entre elas. Valores: 0 a 255.

*PerType*: natureza da tarefa. Valores: p = tarefa periódica
a = tarefa aperiódica
s = tarefa especial

*Status*: estado em que se encontra a tarefa. Valores: "notready", "ready", "running", "waiting".
"notready" = tarefa ainda não foi invocada;
"ready" = tarefa está pronta para executar;
"running" = tarefa está executando;
"waiting = tarefa encontra-se bloqueada;

*RDeadline*: "deadline" relativo. O valor 0 significa que a tarefa não possui prazo para cumprir ("soft deadline").

*Deadline*: "deadline" absoluto.

*Period*: período da tarefa.

*SchNext*: ponteiro para próxima tarefa (BCT) na Fila de Escalonamento (Periódico ou Aperiódico).

*ExecTime*: tempo de execução da tarefa.

*RemExec*: tempo que resta para terminar de executar a tarefa.

*ReadyNext*: ponteiro para próxima tarefa (BCT) na Fila de Pronto.

*Start*: instante em que foi invocada a tarefa.

*Finish*: instante em que foi completada a execução da tarefa.

*Priority*: campo de prioridade disponível. Valores: 0 a 255.

*AuxNext[MAXQ]*:vetor de ponteiro para uma próxima tarefa na respectiva Fila Auxiliar. (MAXQ = nº máximo de Filas Auxiliares).

*Aux1[MAXVARS]*:conjunto de Campos Auxiliares disponíveis no BCT. (Tamanho do campo = 1 byte).

*Aux2[MAXVARS]*:conjunto de Campos Auxiliares disponíveis no BCT. (Tamanho do campo = 4 bytes).

*Command*:ponteiro para uma fila que contém os *comandos genéricos* especificados para a tarefa. A estrutura de dados ComStruct tem o seguinte formato:

```
/* definição do ponteiro ComPtr */
typedef struct ComStruct *ComPtr;
typedef struct ComStruct {
                        unsigned int Type;
                        unsigned int Time;
                        ParPtr       Parameter;
                        ComPtr       ComNext;
                       }
```

onde:

*Type* = número que identifica o comando;
*Time* = instante dentro da tarefa em que ocorre o comando;
*Parameter* = ponteiro para a lista de parâmetros do comando;
*ComNext* = ponteiro para o próximo comando da tarefa.

O campo *PerType* diz respeito ao tipo de tarefa quanto à periodicidade. Quando uma tarefa é inserida, o SSE atribui a este campo o valor "a" ou "p" caso seja uma tarefa aperiódica ou periódica, respectivamente. Para diferenciar um terceiro tipo de tarefa, chamada tarefa especial, foi criado o tipo "s". O significado deste tipo (isto é, da tarefa especial) é particular do algoritmo que o utiliza. Como exemplo, o algoritmo "Deferreable Server" possui uma tarefa denominada servidora que apesar de ser periódica, não é uma tarefa comum. Neste caso, a tarefa servidora foi considerada como uma tarefa especial quando foi implementado o algoritmo (ver Apêndice B).

A tarefa é especificada como especial na função de inicialização (*userinit()*) do Módulo de Escalonamento. Nota-se que a natureza da tarefa especial mantém-se a mesma de início, ou seja, se ela foi inserida como periódica, esta continuará a ser invocada a cada vencimento de seu período.

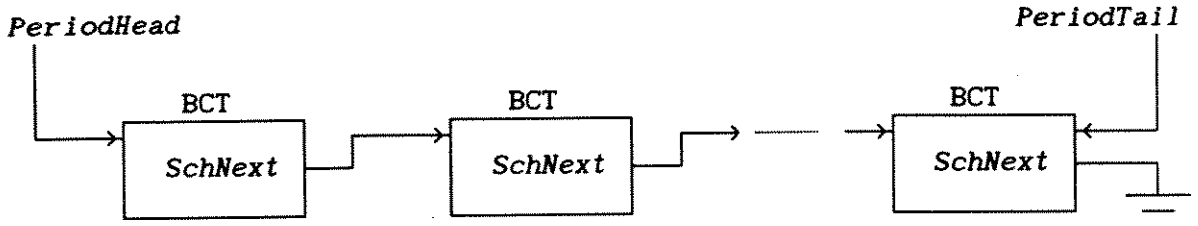
Uma fila de tarefas no SSE é constituida de duas variáveis globais que apontam para o início e fim da fila, e também por um campo do BCT que indica a próxima tarefa na sequência.

Um esquema das filas relacionadas com a interface, mostrando as estruturas de dados, é visto na figura 3.6.

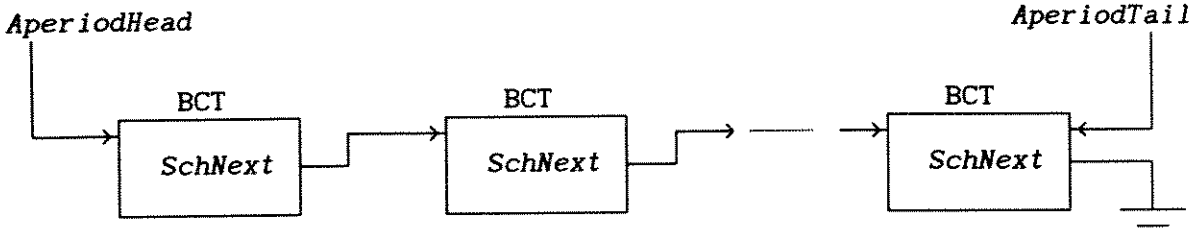
As Filas Auxiliares são filas reservadas para uso particular por parte de algum algoritmo que as requer, não sofrendo qualquer tipo de manipulação pelo SSE.

Com o objetivo de restringir o acesso às estruturas de dados e ao mesmo tempo evitar a necessidade de se conhecer detalhes de implementação do SSE, foi criado um conjunto de serviços que facilitam a utilização da interface SSE - Módulo de Escalonamento para o usuário.

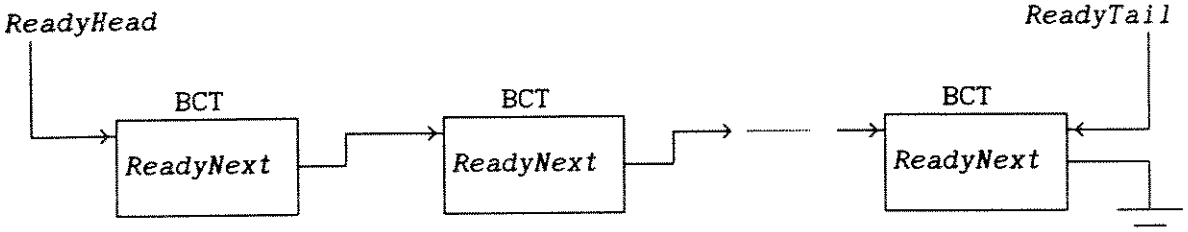
**Fila de Escalonamento Periódico:**




**Fila de Escalonamento Aperiódico:**




**Fila de Pronto:**




**Filas Auxiliares:**

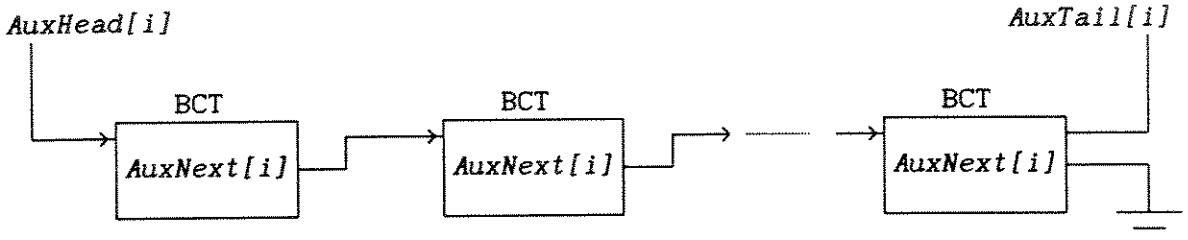



Figura 3.6 - Filas de tarefas do SSE

### 3.3.4 - Serviços de Interface SSE - Módulo de Escalonamento

Os serviços de interface SSE - Módulo de Escalonamento são divididos em três classes:

1 - *Serviços para Manutenção de Filas;*
2 - *Serviços de Acesso aos Parâmetros das Tarefas;*
3 - *Serviços Gerais.*

Qualquer operação, envolvendo as estruturas de dados do SSE, deve ser feita através destes serviços.

Os serviços são implementados por funções C. Para utilizá-los, basta fazer a chamada da respectiva função desejada. A seguir, são descritos os serviços de cada classe, apresentando o nome da função correspondente, sua declaração na linguagem C ("prototype"), descrição, parâmetros e valor de retorno.

#### 3.3.4.1 - Serviços para Manutenção de Filas

As filas de tarefas desempenham papel fundamental na interface SSE - Módulo de Escalonamento. Para cada fila existe um conjunto básico de serviços para sua manipulação, com funções análogas. Os serviços para cada fila são descritos abaixo.

**a) Fila de Escalonamento Periódico:**

**(1) - get_psch_head:**

Declaração C: TaskPtr get_psch_head(void);
Descrição: informa a primeira tarefa da Fila de Escalonamento Periódico;
Valor de Retorno: ponteiro para a primeira tarefa da fila.

**(2) - set_psch_head:**

Declaração C: void set_psch_head(TaskPtr task);

Descrição: atualiza o ponteiro de início da Fila de Escalonamento Periódico;

Parâmetro: task = ponteiro para a tarefa que será a primeira da fila.

**(3) - get_psch_tail:**

Declaração C: TaskPtr get_psch_tail(void);

Descrição: informa a última tarefa da Fila de Escalonamento Periódico;

Valor de Retorno: ponteiro para a última tarefa da fila.

**(4) - set_psch_tail:**

Declaração C: void set_psch_tail(TaskPtr task);

Descrição: atualiza o ponteiro de final da Fila de Escalonamento Periódico;

Parâmetro: task = ponteiro para a tarefa que será a última da fila.

**(5) - get_psch_next:**

Declaração C: TaskPtr get_psch_next(TaskPtr task);

Descrição: informa a tarefa seguinte a uma dada tarefa na Fila Escalonamento Periódico.

Parâmetro: task = ponteiro para a uma dada tarefa.

Valor de Retorno: ponteiro para a tarefa seguinte à tarefa especificada.

**(6) - set_psch_next:**

Declaração C: void set_psch_next(TaskPtr task1, TaskPtr task2);

Descrição: insere uma tarefa na posição seguinte a uma dada tarefa, na Fila de Escalonamento Periódico.

Parâmetros: task1 = ponteiro para uma dada tarefa;
task2 = ponteiro para a tarefa a ser inserida.

Observação: esta função não faz a inserção completa de uma tarefa na fila, pois ela não realiza a ligação da tarefa inserida com a que se encontrava seguinte a task1 (figura 3.7).

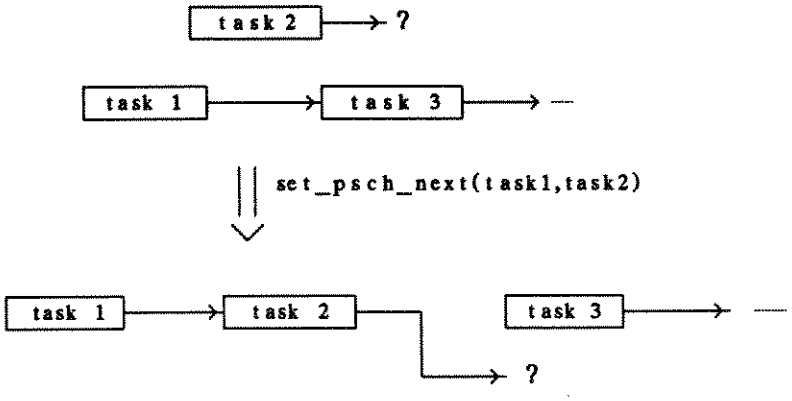



Figura 3.7 - Esquema de inserção da função set_psch_next()

A justificativa para não se ter implementado uma inserção completa, foi para manter a função flexível e servir de base para construção de outras funções mais elaboradas. A inserção completa é realizada pela função *insert_pqueue()*.

**(7) - insert_pqueue:**

Declaração C: void insert_pqueue(TaskPtr task1, TaskPtr task2);

Descrição: insere uma tarefa na Fila de Escalonamento Periódico após uma dada tarefa.

Parâmetros: task1 = ponteiro para a tarefa depois da qual deseja-se inserir a nova tarefa;
task2 = ponteiro para a tarefa a ser inserida.

Observação: esta função realiza a inserção completa da tarefa na fila como mostra a figura 3.8.

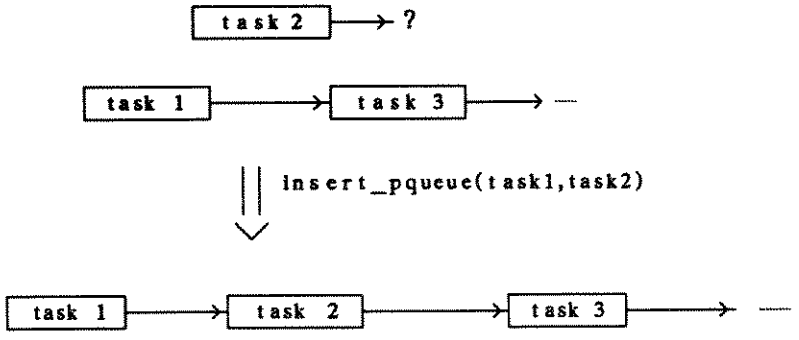



Figura 3.8 - Esquema de inserção da função insert_pqueue()

**b) Fila de Escalonamento Aperiódico:**

**(8) - get_asch_head:**

Declaração C: TaskPtr get_asch_head(void);

Descrição: informa a primeira tarefa da Fila de Escalonamento Aperiódico.

Valor de Retorno: ponteiro para a primeira tarefa da fila.

**(9) - set_asch_head:**

Declaração C: void set_asch_head(TaskPtr task);

Descrição: atualiza o ponteiro de início da Fila de Escalonamento Aperiódico.

Parâmetro: task = ponteiro para tarefa que será a primeira da fila.

**(10) - get_asch_tail:**

Declaração C: TaskPtr get_asch_tail(void);

Descrição: informa a última tarefa da Fila de Escalonamento Aperiódico.

Valor de Retorno: ponteiro para a última tarefa da fila.

**(11) - set_asch_tail:**

Declaração C: void set_asch_tail(TaskPtr task);

Descrição: atualiza o ponteiro de final da Fila de Escalonamento Aperiódico.

Parâmetro: task = ponteiro para tarefa que será a última da fila.

**(12) - get_asch_next:**

Declaração C: TaskPtr get_asch_next(TaskPtr task);

Descrição: informa a tarefa seguinte a uma dada tarefa, na Fila de Escalonamento Aperiódico.

Parâmetro: task = ponteiro para a uma tarefa.

Valor de Retorno: ponteiro para a tarefa seguinte a especificada.

**(13) - set_asch_next:**

Declaração C: void set_asch_next(TaskPtr task1, TaskPtr task2);

Descrição: insere uma tarefa na posição seguinte a uma dada tarefa, na Fila de Escalonamento Aperiódico.

Parâmetros: task1 = ponteiro para uma dada tarefa;
task2 = ponteiro para a tarefa a ser inserida.

Observação: a mesma para a função *set_psch_next()*.

**(14) - insert_aqueue:**

Declaração C: void insert_aqueue(TaskPtr task1, TaskPtr task2);

Descrição: insere uma tarefa na Fila de Escalonamento Aperiódico após uma dada tarefa.

Parâmetros: task1 = ponteiro para a tarefa a qual se deseja inserir após;
task2 = ponteiro para a tarefa a ser inserida.

**c) Fila de Pronto:**

**(15) - get_ready_head:**

Declaração C: TaskPtr get_ready_head(void);

Descrição: informa a primeira tarefa da Fila de Pronto.

Valor de Retorno: ponteiro para a primeira tarefa da fila.

**(16) - set_ready_head:**

Declaração C: void set_ready_head(TaskPtr task);

Descrição: atualiza o ponteiro de início da Fila de Pronto.

Parâmetro: task = ponteiro para a tarefa que será a primeira da fila.

**(17) - get_ready_tail:**

Declaração C: TaskPtr get_ready_tail(void);

Descrição: informa a última tarefa da Fila de Pronto.

Valor de Retorno: ponteiro para a última tarefa da fila.

**(18) - set_ready_tail:**

Declaração C: void set_ready_tail(TaskPtr task);

Descrição: atualiza o ponteiro de final da Fila de Pronto.

Parâmetro: task = ponteiro para tarefa que será a última da fila.

**(19) - get_ready_next:**

Declaração C: TaskPtr get_ready_next(TaskPtr task);

Descrição: informa a tarefa seguinte a uma dada tarefa, na Fila de Pronto.

Parâmetro: task = ponteiro para a uma tarefa.

Valor de Retorno: ponteiro para a tarefa seguinte a especificada.

**(20) - set_ready_next:**

Declaração C: void set_ready_next(TaskPtr task1, TaskPtr task2);

Descrição: insere uma tarefa na posição seguinte a uma dada tarefa, na Fila de Pronto.

Parâmetros: task1 = ponteiro para uma dada tarefa;
task2 = ponteiro para a tarefa a ser inserida.

Observação: a mesma para a função *set_psch_next()*.

**(21) - insert_readyqueue:**

Declaração C: void insert_readyqueue(TaskPtr task1, TaskPtr task2);
Descrição: insere uma tarefa na Fila de Pronto após uma dada tarefa.
Parâmetros: task1 = ponteiro para a tarefa a qual se deseja inserir após;
task2 = ponteiro para a tarefa a ser inserida.

**d) Filas Auxiliares:**

**(22) - get_aux_head:**

Declaração C: TaskPtr get_aux_head(int queue);
Descrição: informa a primeira tarefa da Fila Auxiliar especificada.
Parâmetro: queue = nº da Fila Auxiliar que se está trabalhando.
Valor de Retorno: ponteiro para a primeira tarefa da fila.

**(23) - set_aux_head:**

Declaração C: void set_aux_head(int queue, TaskPtr task);
Descrição: atualiza o ponteiro de início da Fila Auxiliar especificada.
Parâmetro: queue = nº da Fila Auxiliar que se está trabalhando.
task = ponteiro para tarefa que será a primeira da fila.

**(24) - get_aux_tail:**

Declaração C: TaskPtr get_aux_tail(int queue);
Descrição: informa a última tarefa da Fila Auxiliar especificada.
Parâmetro: queue = nº da Fila Auxiliar que se está trabalhando.
Valor de Retorno: ponteiro para a última tarefa da fila.

**(25) - set_aux_tail:**

Declaração C: void set_aux_tail(int queue, TaskPtr task);

Descrição: atualiza o ponteiro de final da Fila Auxiliar especificada.

Parâmetro: queue = nº da Fila Auxiliar que se está trabalhando.
task = ponteiro para a tarefa que será a última da fila.

**(26) - get_aux_next:**

Declaração C: TaskPtr get_aux_next(int queue, TaskPtr task);

Descrição: informa a tarefa seguinte a uma dada tarefa, na Fila Auxiliar especificada.

Parâmetro: queue = nº da Fila Auxiliar que se está trabalhando.
task = ponteiro para uma dada tarefa.

Valor de Retorno: ponteiro para a tarefa seguinte a especificada.

**(27) - set_aux_next:**

Declaração C: void set_aux_next(int queue, TaskPtr task1,
TaskPtr task2);

Descrição: insere uma tarefa na posição seguinte a uma dada tarefa, na Fila Auxiliar especificada.

Parâmetros: queue = nº da Fila Auxiliar que se está trabalhando
task1 = ponteiro para uma dada tarefa;
task2 = ponteiro para a tarefa a ser inserida.

Observação: a mesma para a função *set_psch_next()*.

**(28) - insert_auxqueue:**

Declaração C: void insert_auxqueue(int queue, TaskPtr task1, TaskPtr task2);

Descrição: insere uma tarefa na Fila Auxiliar especificada, após uma dada tarefa.

Parâmetros: queue = fila auxiliar que se está trabalhando;
task1 = ponteiro para a tarefa a qual se deseja inserir após;
task2 = ponteiro para a tarefa a ser inserida.

### 3.3.4.2 - Serviços de Acesso aos Parâmetros das Tarefas

**(29) - get_atask_adrs:**

Declaração C: TaskPtr get_atask_adrs(unsigned char tid);

Descrição: informa o ponteiro para a tarefa aperiódica especificada por seu respectivo identificador.

Parâmetro: tid = identificador da tarefa.

Valor de Retorno: ponteiro para a tarefa aperiódica especificada.

**(30) - get_ptask_adrs:**

Declaração C: TaskPtr get_ptask_adrs(unsigned char tid);

Decrição: informa o ponteiro para a tarefa periódica especificada por seu respectivo identificador.

Parâmetro: tid = identificador da tarefa.

Valor de Retorno: ponteiro para a tarefa periódica especificada.

**(31) - get_task_id:**

Declaração C: usigned char get_task_id(TaskPtr task);

Descrição: informa o identificador da tarefa dado o seu ponteiro. A natureza da tarefa (periódica, aperiódica ou especial) é indiferente para esta função.

Parâmetro: task = ponteiro para a tarefa.

Valor de Retorno: identificador da tarefa.

**(32) - get_deadline:**

Declaração C: unsigned long get_deadline(TaskPtr task);

Descrição: informa o "deadline" absoluto de uma tarefa.

Parâmetro: task = ponteiro para a tarefa.

Valor de Retorno: "deadline" absoluto da tarefa.

**(33) - set_deadline:**

Declaração C: void set_deadline(TaskPtr task, unsigned long d);

Descrição: atribui novo valor de "deadline" absoluto à tarefa.

Parâmetros: task = ponteiro para a tarefa.

d = "deadline" relativo. Exemplo: "tick" atual = 10, "deadline" absoluto desejado = 15, então: "deadline" relativo = 15 - 10 = 5.

**(34) - get_r_deadline:**

Declaração C: unsigned long get_r_deadline(TaskPtr task);

Descrição: informa o "deadline" relativo de uma tarefa.

Parâmetro: task = ponteiro para a tarefa.

Valor de Retorno: "deadline" relativo da tarefa.

**(35) - set_r_deadline:**

Declaração C: void set_r_deadline(TaskPtr task, unsigned long d);

Descrição: atribui novo valor ao "deadline" relativo de uma tarefa.

Parâmetros: task = ponteiro para a tarefa

d = "deadline" relativo.

**(36) - get_exec_time:**

Declaração C: unsigned long get_exec_time(TaskPtr task);

Descrição: informa o tempo de execução de uma tarefa.

Parâmetro: task = ponteiro para a tarefa.

Valor de Retorno: tempo de execução da tarefa.

**(37) - get_start_time:**

Declaração C: unsigned long get_start_time(TaskPtr task);

Descrição: informa o instante de invocação de uma tarefa.

Parâmetro: task = ponteiro para a tarefa.

Valor de Retorno: instante de invocação da tarefa.

**(38) - get_finish_time:**

Declaração C: unsigned long get_finish_time(TaskPtr task);

Descrição: informa o instante que a execução de uma tarefa foi completada.

Parâmetro: task = ponteiro para a tarefa.

Valor de Retorno: instante final de execução da tarefa.

**(39) - get_period:**

Declaração C: unsigned long get_period(TaskPtr task);

Descrição: informa o período de uma tarefa.

Parâmetro: task = ponteiro para a tarefa.

Valor de Retorno: período da tarefa.

**(40) - set_period:**

Declaração C: void set_period(TaskPtr task, unsigned long p);

Descrição: atribui novo período à tarefa.

Parâmetro: task = ponteiro para a tarefa
p = período.

**(41) - get_priority:**

Declaração C: unsigned char get_priority(TaskPtr task);

Descrição: informa a prioridade de uma tarefa.

Parâmetro: task = ponteiro para tarefa.

Valor de Retorno: prioridade da tarefa. O valor informado é o mesmo para o campo Prioridade, quando a tarefa é inserida.

**(42) - set_priority:**

Declaraçao C: void set_priority(TaskPtr task, unsigned char pr);

Descrição: atribui um valor de prioridade à tarefa.

Parâmetros: task = ponteiro para tarefa
pr = prioridade.

**(43) - get_aux1_var:**

Declaração C: unsigned char get_aux1_var(TaskPtr task, unsigned int variable);

Descrição: informa o valor de uma variável auxiliar especificada, pertencente a uma dada tarefa. A variável é do tipo unsigned char (1 byte).

Parâmetro: task = ponteiro para a tarefa a qual pertence a variável;
variable = especifica a variável que se está trabalhando. Valores entre 0 e MAXVAR-1 (definido no apêndice A).

Valor de Retorno: valor da variável auxiliar especificada.

**(44) - set_aux1_var:**

Declaração C: void set_aux1_var(TaskPtr task, unsigned int variable,
unsigned char value);

Descrição: atribui valor à uma variável auxiliar especificada, pertencente a uma dada tarefa. A variável é do tipo unsigned char (1 byte).

Parâmetro: task = ponteiro para a tarefa a qual pertence a variável;
variable = especifica a variável que se está trabalhando. Valores entre 0 e MAXVAR-1 (definido no apêndice A);
value = valor a ser atribuido à variável (entre 0 e 255).

**(45) - get_aux2_var:**

Declaração C: unsigned char get_aux2_var(TaskPtr task, unsigned int variable);

Descrição informa o valor de uma variável auxiliar especificada, pertencente a uma da tarefa. A variável é do tipo unsigned char (4 byte).

Parâmetro: task = ponteiro para a tarefa a qual pertence a variável;
variable = especifica a variável que se está trabalhando. Valores entre 0 e MAXVAR-1 (definido no apêndice A).

Valor de Retorno: valor da variável.

**(46) - set_aux2_var:**

Declaração C: void set_aux2_var(TaskPtr task, unsigned int variable,
unsigned char value);

Descrição: atribui valor à uma variável auxiliar especificada, pertencente a uma dada tarefa. A variável é do tipo unsigned char (4 byte).

Parâmetro: task = tarefa a qual pertence a variável;
variable = especifica a variável que se está trabalhando. Valores entre 0 e MAXVAR-1 (defi nido no apêndice A);
value = valor a ser atribuido à variável.

**(47) - get_rem_exec:**

Declaração C: unsigned long get_rem_exec(TaskPtr task);

Descrição: informa o tempo restante de execução de uma tarefa.

Parâmetro: task = ponteiro para tarefa.

Valor de Retorno: tempo restante de execução da tarefa.

**(48) - get_task_status:**

Declaração C: enum TSt get_task_status(TaskPtr task);

Descrição: informa o estado que se encontra a tarefa.

Parâmetro: task = ponteiro para uma dada tarefa.

Valor de Retorno: estado da tarefa (*notready*, *ready*, *running*, *waiting*).

**(49) - set_task_status:**

Declaração C: void set_task_status(TaskPtr task, enum TSt ts);

Descrição: atribui novo status à tarefa.

Parâmetro: task = ponteiro para uma dada tarefa
ts = status.

### 3.3.4.3 - Serviços Gerais

**(50) - get_time:**

Declaração C: unsigned long get_time(void);

Descrição: informa o instante atual da simulação.

Valor de Retorno: instante atual da simulação.

**(51) - get_cap_service:**

Declaração C: unsigned long get_cap_service(TaskPtr task);

Descrição: informa a capacidade de serviço de uma tarefa servidora.

Parâmetro: task = ponteiro para tarefa servidora.

Valor de Retorno: capacidade de serviço da tarefa.

Observação: os conceitos de capacidade de serviço e tarefa servidora são discutidos no cpítulo II na apresentação dos algoritmos de *Taxa Monotônica* considerando tarefas aperiódicas.

**(52) - set_cap_service:**

Declaração C: void set_cap_service(TaskPtr task, unsigned long cs);

Descrição: atualiza a capacidade de serviço de uma tarefa servidora.

Parâmetros: task = ponteiro para tarefa servidora
cs = capacidade de serviço.

**(53) - get_laxity:**

Declaração C: long get_laxity(TaskPtr task);

Descrição: informa a folga de execução de uma tarefa.

Parâmetro: task = ponteiro para tarefa.

Valor de Retorno: folga da tarefa.

**(54) - get_running:**

Declaração C: TaskPtr get_running(void);

Descrição: informa a tarefa que está executando.

Valor de Retorno: ponteiro para tarefa executando no instante atual.

**(55) - get_last_running:**

Declaração C: TaskPtr get_last_running(void);

Descrição: informa a tarefa que estava executando na última vez que foi invocado o escalonador.

Valor de Retorno: ponteiro para a tarefa que se encontrava executando no instante da última invocação do escalonador.

**(56) - set_sched_control_userint:**

Declaraçao C: void set_sched_control_userint(void);

Descrição: indica ao sistema que o controle de invocação do escalonador será feito através da função *userinter()*.

**(57) - set_sched_control_SSE:**

Declaração C: void set_sched_control_SSE(void);

Descrição: indica ao sistema que o controle de invocação do escalonador será feito pelo SSE.

**(58) - get_status_command:**

Declaração C: TaskPtr get_status_command(void);

Descrição: indica se foi executado algum *comando genérico* no instante atual da simulação.

Valor de Retorno: ponteiro para a tarefa que executou o *comando genérico*. Se o valor for NULL (ponteiro nulo) significa que não foi executado um comando naquele instante.

**(59) - get_type_command:**

Declaração C: unsigned int get_type_command(void);

Descrição: indica o tipo de *comando genérico* executando no instante atual da simulação.

Valor de Retorno: tipo de comando. O valor 0 significa que não foi executado um comando naquele instante.

**(60) - get_parameter:**

Declaração C: int get_parameter(unsigned nr);

Descrição: informa o valor de um parâmetro específico de um *comando genérico* executado.

Parâmetro: nr = indica qual o parâmetro que se deseja saber o valor. Por exemplo, para atribuir o valor do segundo parâmetro de um *comando genérico* à uma variável *x*:

```
x = get_parameter(2);
```

Valor de Retorno: o valor do parâmetro especificado.

Nesta seção foram apresentados os serviços disponíveis à implementação de políticas de escalonamento. A seguir, a utilização destes serviços será mostrada através de dois exemplos de escalonadores presentes no SSE.

### 3.3.5 - Exemplos de Implementação de Políticas de Escalonamento

O objetivo desta seção é ilustrar a construção do Módulo de Escalonamento através da apresentação de dois exemplos de políticas implementadas no SSE. Os exemplos escolhidos ilustram:

1 - como deve ser a estrutura funcional do Módulo de Escalonamento;
2 - a diferença entre o controle de invocação do escalonador feito pelo SSE e pelo Módulo de Escalonamento;
3 - a utilização de serviços disponíveis para a implementação do escalonador.

#### 3.3.5.1 - Exemplo I - "Earliest Deadline"

A política de escalonamento *"Earliest Deadline"* prioriza as tarefas cujo "deadline" está mais próximo, ou seja, a tarefa (pronta) que possui o menor prazo para cumprir o seu processamento, tem direito a ocupar a CPU.

Na implementação do módulo, o primeiro passo é determinar quem fará o controle de invocação do escalonador. Na política *"Earliest Deadline"*, os instantes de escalonamento ocorrem a cada chegada de tarefa. Como este evento é reconhecido pelo SSE, o controle de invocação do escalonador pode ser feito por ele, o que será indicado na rotina de inicialização (*userinit()*). Neste caso, a rotina de interrupção do usuário (*userinter()*) não terá instrução alguma, pois não será usada.

O algoritmo do escalonador é bem simples. Quando alguma tarefa é invocada, o SSE insere-a na Fila de Escalonamento correspondente (Periódico ou Aperiódico) e chama o escalonador (*scheduler()*). Por sua vez, o escalonador retira as tarefas das Filas de Escalonamento e as insere na Fila de Pronto, observando a ordem por "deadline" destas.

O Módulo de Escalonamento constituido pelas três funções C descritas, é apresentado abaixo.

```
void userinit(void)
{
  /* Indica que o controle de invocação do escalonador será feito pelo SSE */
  set_sched_control_SSE();
}


Boolean userinter(void)
{
  /* Este módulo não tem qualquer efeito na implementação da
  ** política "earliest deadline", devendo constar apenas por
  ** compatibilidade com a interface definida
  */
  return(true);
}


void scheduler(void)

{
  /* Esta função é invocada a cada chegada de tarefa nas filas de
  ** escalonamento.
  ** Variáveis:
  **            SchDeadline = deadline absoluto da tarefa a ser
```

```
**                          inserida na Fila de Pronto;
**              SHead = ponteiro para a primeira tarefa na Fila de
**                      Escalonamento;
**              RHead =ponteiro para a primeira tarefa na Fila de
**                      Pronto;
**              TPtr1, TPtr2 = ponteiros auxiliares para tarefas.
**              queue = Fila de Escalonamento que se está traba-
**                      lhando.   queue 1 = Fila Periódica de Esc.
**                                queue 2 = Fila Aperiódica de Esc.
**   Notação:
**              FEP = Fila de Escalonamento Periódico
**              FEA = Fila de Escalonamento Aperiódico
**              FP  = Fila de Pronto
**                                                              */

unsigned long SchDeadline;
TaskPtr SHead, RHead, TPtr1, TPtr2; /* ponteiros para tarefa    */
int queue;

for (queue =  1; queue <= 2; queue++)
  {
    /* Se a fila de trabalho é a FEP                            */
    if (queue == 1)
      /* pega tarefa periódica a ser escalonada                 */
     SHead = get_psch_head();
      /* senão trabalha com a FEA                               */
      else
       /* pega tarefa aperiódica a ser escalonada               */
      SHead = get_asch_head();
    /* Enquanto há tarefa p/ escalonar                          */
   while (SHead != NULL)
      {
        /* pega a 1ª tarefa da FP                               */
        RHead = get_ready_head();
```

```
/* Se FP não tem tarefas                                    */
if (RHead == NULL)
  {
   /* insere a tarefa na FP                                  */
    set_ready_head(SHead);
    set_ready_next(SHead,NULL);
  }
/* se há tarefas na FP                                      */
else
  {
    /* lê o deadline da tarefa a ser escalonada             */
    SchDeadline = get_deadline(SHead);
    /* Se o deadline for menor que o da 1ª tarefa da FP     */
    if (SchDeadline < get_deadline(RHead))
      {
        /* insere a tarefa em 1º na FP                      */
        set_ready_next(SHead,RHead);
        /* atualiza a cabeça da FP                          */
        set_ready_head(SHead);
      }
    else                    /* senão                        */
      {
        /* encontra a posição adequada na FP                */
        TPr2 = get_ready_head();
        while ((TPr2 != NULL) &&
               (SchDeadline >= get_deadline(TPtr2)))
          {
            TPtr1 = TPtr2;
            TPtr2 = get_ready_next(TPtr2);
          }
        /* e insere a tarefa na FP                          */
        set_ready_next(TPtr1,SHead);
        set_ready_next(SHead,TPtr2);
      }
```

```
            }
          /* Se fila de trabalho é a FEP                          */
          if (queue == 1)
            {
              /* pega próxima tarefa periódica p/ escalonar       */
              SHead = get_psch_next(SHead);
              /* e atualiza a cabeça da FEP                       */
              set_psch_head(SHead);
            }
          else /* Se a fila de trabalho é a FEA                   */
            {
              /* pega próxima tarefa aperiódica p/ escalonar      */
              SHead = get_asch_next(SHead);
              /* e atualiza a cabeça da FEA                       */
               set_asch_head(SHead);
            }
        }
    }
}
```

### 3.3.5.2 - Exemplo II - "Least Laxity"

O escalonamento *"Least Laxity"* prioriza tarefas que possuem menor folga de tempo para terminar de executar sem que ultrapasse seu deadline. Nesta política, enquanto uma tarefa está executando, a sua folga mantém-se constante. Já o mesmo não ocorre com a folga das tarefas que aguardam a vez. Pode acontecer que, em um certo instante, a tarefa rodando tenha mais folga do que a tarefa que se encontra na Fila de Pronto, o que deve ocasionar necessariamente um escalonamento. Este fato exemplifica um tipo de evento não reconhecido pelo SSE, o que sugere que o controle de invocação do escalonador seja feito pela interrupção do usuário (*userinter()*). Dois eventos devem ser checados nesta rotina: (1) chegada de tarefas (periódicas e aperiódicas) e (2) alteração da prioridade da tarefa executando devido à mudança da folga das tarefas prontas. Na ocorrência de um dos eventos, o escalonador deve ser invocado pa-

ra atualizar a Fila de Pronto. O Módulo de Escalonamento, neste exemplo, é mais elaborado que o anterior, por causa do tratamento a ser dado para os dois eventos possíveis de ocorrerem. No caso do primeiro evento, a tarefa é retirada da respectiva Fila de Escalonamento e inserida na Fila de Pronto, observando o esquema de prioridade. Já para o segundo tipo de evento, a Fila de Pronto é reordenada para atualizar a prioridade de execução das tarefas.

As três funções C que constituem o Módulo de Escalonamento referente à política "Least Laxity" são apresentadas abaixo.

```
void userinit(void)
{
  /* Indica que o controle de invocação do escalonador será feito
  ** pela rotina de interrupção do usuario (userinter()).          */

  set_sched_control_userint();
}


Boolean userinter(void)
{
  /* Esta função sinaliza uma operação de escalonamento caso a
  ** folga da tarefa que está executando tenha ficado maior do que
  ** a folga de alguma tarefa pronta, ou exista tarefa para
  ** escalonar.                                                    */
  TaskPtr RHead, TPtr;

  /* pega a tarefa que está executando (1ª na FP) . . .           */
  RHead = get_ready_head();
  /*                                    se houver alguma          */
  if (RHead != NULL)
    {
    /* pega a tarefa seguinte na FP                               */
      TPtr = get_ready_next(RHead);
```

```
      /* Se houver tarefa seguinte e sua folga for menor que a da     */
      /* tarefa que está executando                                   */
      if ((TPtr!=NULL) &&
          (get_laxity(RHead) > get_laxity(TPtr)))
        /* invoca o escalonador                                       */
        return(true);
    }
  /* Obs.: basta verificar a folga da tarefa seguinte à cabeça da
  ** FP, pois esta fila encontra-se organizada em ordem crescente
  ** de folga da tarefa                                              */
  /* Se houver tarefas nas filas de escalonamento FEP ou FEA         */
  if ((get_psch_head() != NULL) || (get_asch_head() != NULL))
    /* invoca o escalonador                                         */
    return(true);
  /* senão                                                          */
  else
    /* nada feito !                                                 */
    return(false);
}
void scheduler(void)
{
/* Esta função é invocada quando:
**    1 - uma tarefa na FP, que aguarda a vez de executar, fica com a
**        folga menor que a da tarefa que está "rodando";
**    2 - houver tarefas nas filas de escalonamento FEP ou FEA.
**
**    Variáveis:
**            SchLaxity = folga da tarefa a ser escalonada
**            SHead = ponteiro para a primeira tarefa na Fila de
**                    Escalonamento;
**            RHead =ponteiro para a primeira tarefa na Fila de
**                    Pronto;
**            TPtr1, TPtr2 = ponteiros auxiliares para tarefas.
**            queue = Fila de Escalonamento que se está traba-
```

```
**                    lhando.  queue 1 = Fila de Esc. Periódico;
**                             queue 2 = Fila de Esc. Aperiódico.
**
**    Notação:
**          FEP = Fila de Escalonamento Periódico
**          FEA = Fila de Escalonamento Aperiódico
**          FP  = Fila de Pronto
**                                                              */
  long SchLaxity;
  TaskPtr TPtr1, TPtr2, SHead, RHead;
  int queue;

  /* pega primeira tarefa na FP                                 */
  RHead = get_ready_head();
  /* Se há tarefa executando                                    */
if (RHead != NULL)
    {
      /* verifica se houve alteração de prioridade devido a variação
      ** da folga das tarefas                                   */
      SchLaxity = get_laxity(RHead);
      TPtr2 = get_ready_next(RHead);
      while ((TPtr2 != NULL) &&
             (SchLaxity > get_laxity(TPtr2)))
        {
          TPtr1 = TPtr2;
          TPtr2 = get_ready_next(TPtr2);
        }
      /* Se alterou a tarefa de maior prioridade                */
      if (TPtr2 != get_ready_next(RHead))
        {
          /* reordena a FP                                      */
          set_ready_head(get_ready_next(RHead));
          set_ready_next(TPtr1,RHead);
          set_ready_next(RHead,TPtr2);
```

```
        }
    }
for (queue = 1; queue <= 2; queue++)
    {
      /* Se a fila de trabalho é a FEP                          */
      if (queue == 1)
        /* pega tarefa periódica a ser escalonada               */
        SHead = get_psch_head();
      /* senão trabalha com a FEA                               */
      else
        /* pega tarefa aperiódica a ser escalonada              */
        SHead = get_asch_head();
      /* Enquanto há tarefa p/ escalonar                        */
      while (SHead != NULL)
        {
          /* pega a 1ª tarefa da FP                           */
          RHead = get_ready_head();
          /* Se FP não tem tarefas                            */
          if (RHead == NULL)
            {
             /* insere a tarefa na FP                         */
              set_ready_head(SHead);
              set_ready_next(SHead,NULL);
            }
          /* se há tarefas na FP                              */
          else
            {
             /* lê a folga da tarefa a ser escalonada         */
              SchLaxity = get_laxity(SHead);
              /* Se a folga for menor que o da 1ª tarefa da FP */
               if (SchLaxity < get_laxity(RHead))
                {
                   /* insere a tarefa em 1º na FP           */
                    set_ready_next(SHead,RHead);
```

```
            /* atualiza a cabeça da FP                          */
            set_ready_head(SHead);
          }
        else                        /* senão                    */
          {
            /* encontra a posição adequada na FP                */
            TPr2 = get_ready_head();
            while ((TPr2 != NULL) &&
                   (SchLaxity >= get_laxity(TPtr2)))
              {
                TPtr1 = TPtr2;
                TPtr2 = get_ready_next(TPtr2);
              }
            /* e insere a tarefa na FP                          */
            set_ready_next(TPtr1,SHead);
            set_ready_next(SHead,TPtr2);
          }
      }
    /* Se fila de trabalho é a FEP                              */
    if (queue == 1)
      {
        /* pega próxima tarefa periódica p/ escalonar           */
        SHead = get_psch_next(SHead);
        /* e atualiza a cabeça da FEP                           */
        set_psch_head(SHead);
      }
    /* Se a fila de trabalho é a FEA                            */
    else
      {
        /* pega próxima tarefa aperiódica p/ escalonar          */
        SHead = get_asch_next(SHead);
        /* e atualiza a cabeça da FEA                           */
        set_asch_head(SHead);
      }
```

```
        }
    }
}
```

### 3.3.6 - Recursos Flexíveis do SSE

O SSE conta com um conjunto de estruturas de dados e serviços disponíveis ao usuário. A razão da existência destes recursos é fornecer suporte e flexibilidade para a implementação de diferentes tipos de Módulo de Escalonamento.

No decorrer do capítulo foram apresentados estes recursos dentro do contexto em que se encontravam. Esta seção reune tais recursos, mostrando maiores detalhes de como utilizá-los. Um exemplo ao final ilustrará a aplicação de alguns deles.

#### 3.3.6.1 - Parâmetro Prioridade

Quando uma tarefa é inserida no sistema, entre os vários parâmetros que devem ser preenchidos, existe um denominado Prioridade. Este parâmentro encontra-se à disposição do usuário para ser empregado em algum esquema de prioridade particular do algoritmo de escalonamento, o qual se está implementando. Entre os escalonadores internos do SSE, o parâmetro Prioridade é utilizado apenas pelo algoritmo de *Prioridade Fixa Simples*, pois para os demais, este campo não tem efeito, devido a cada um possuir um critério de prioridade próprio.

Existem duas funções para acessar o parâmetro Prioridade que podem ser utilizadas no Módulo de Escalonamento:

1 - *get_priority()*: retorna o valor do campo Prioridade da tarefa;

2 - *set_priority()*: atribui um valor ao campo Prioridade da tarefa.

A descrição completa destas funções (nº 41 e nº 42) é feita na seção 3.3.4.2.

### 3.3.6.2 - Filas Auxiliares

Na implementação de determinadas funções do Módulo de Escalonamento, pode ser necessário trabalhar com fila(s) de tarefas além daquelas normalmente implementadas no sistema. No SSE, foi criado um conjunto de filas, chamadas Filas Auxiliares, para serem manipuladas da forma que o usuário achar adequada.

Sete funções da interface SSE - Módulo de Escalonamento dão suporte a operação destas filas:

1 - *get_aux_head()*: informa a primeira tarefa de uma Fila Auxiliar;

2 - *set_aux_head()*: atualiza o ponteiro de início de uma Fila Auxiliar;

3 - *get_aux_tail()*: informa a última tarefa de uma Fila Auxiliar;

4 - *set_aux_tail()*: atualiza o ponteiro de final de uma Fila Auxiliar;

5 - *get_aux_next()*: informa qual a tarefa seguinte a uma dada tarefa de uma Fila Auxiliar;

6 - *set_aux_next()*: atualiza a tarefa seguinte a uma dada tarefa de uma Fila Auxiliar.

7 - *insert_auxqueue()*: insere uma tarefa em uma Fila Auxiliar.

A descrição completa destas funções (nº 22 a nº 28) é vista na seção 3.3.4.1. O número máximo de Filas Auxiliares que o SSE suporta, é especificado no apêndice A.

### 3.3.6.3 - Campos Auxiliares

Ao ser definido o BCT de uma tarefa, foram reservados campos de dados para uso geral. Qualquer função que precise de variáveis associadas às tarefas (BCT), pode utilizar estes campos auxiliares.

Existem dois conjuntos de campos auxiliares que são diferenciados pelo tipo de dados. O primeiro possui variáveis do tipo *unsigned char* (1 byte), e o outro, variáveis do tipo *unsigned long* (4 byte). A escolha de qual conjunto trabalhar é dependente da ordem de grandeza dos valores a serem operados.

O acesso a estes campos é realizado através das funções:

1 - *get_aux1_var()*: informa o valor de uma variável especificada do tipo *unsigned char*;

2 - *set_aux1_var()*: atribui um valor à uma variável especificada do tipo *unsigned char*;

3 - *get_aux2_var()*: idem a *get_aux1_var()* com a diferença do tipo que é *unsigned long*;

4 - *set_aux2_var()*: idem a *set_aux1_var()* com a diferença do tipo que é *unsigned long*.

Uma descrição detalhada destas funções (nº 43 a nº 46, respectivamente) é vista na seção 3.3.4.2. O número máximo de Campos Auxiliares de cada tipo é encontrado no apêndice A.

### 3.3.6.4 - Comandos Genéricos

Foi visto na seção 3.2.1 que é possível definir *comandos genéricos* dentro de uma tarefa, informando-se para cada um, o tipo, o instante de ocorrência na tarefa e os parâmetros que possui (se houver algum). O objetivo é permitir ao usuário criar comandos que possam resultar num escalonamento das tarefas da simulação. Como o próprio nome sugere, o *comando genérico* não realiza uma função específica. Quando é atingido o instante de sua ocorrência, acontece uma preempção no processamento da tarefa que o está "executando" e o Módulo de Escalonamento é invocado. Este, por sua vez, deve realizar a interpretação e o tratamento do *comando genérico*. O SSE é responsável apenas por informar a ocorrência do evento.

Vários tipos de comandos podem ser simulados através deste recurso como, por exemplo, operadores de semáforos, comando *wait(time)* (bloqueia a execução da tarefa por *time* "ticks"), etc. Também pode-se associar funções particulares à ocorrência dos *comandos genéricos*.

O parâmetro *Tipo* do *comando genérico* serve para identificar um comando particular, e assim, para cada um que se queira implementar, deve ser associado um respectivo tipo.

O procedimento para trabalhar com *comandos genéricos* é o seguinte:

1 - Verificar se ocorreu algum comando em determinado instante;

2 - Identificar a tarefa que "executou" o comando;

3 - Identificar o tipo de *comando genérico*;

4 - Ler o(s) parâmetro(s), se houver;

5 - Tomar as ações adequadas, conforme o tipo de *comando genérico*.

Os passos de 1 a 4 são realizados através de três funções presentes nos serviços de interface SSE - Módulo de Escalonamento:

1 - *get_status_command()*: indica se foi executado algum *comando genérico* no instante atual da simulação. Caso positivo, é retornado o ponteiro para a tarefa que executou o comando, do contrário, um valor nulo é informado;

2 - *get_type_command()*: retorna o tipo do *comando genérico* executado no instante atual da simulação. O valor 0 significa que não ocorreu um comando naquele instante.

3 - *get_parameter()*: informa o valor de um parâmetro específico de um *comando genérico* executado.

A descrição das funções (nº 58 a nº 60) encontra-se detalhada na seção 3.3.4.3.

O exemplo a seguir mostra a aplicação de alguns dos recursos vistos até agora.

### 3.3.6.5 - Exemplo de Aplicação dos Recursos Flexíveis

Foi escolhido para ilustrar o uso dos recursos flexíveis do SSE, a implementação do comando *wait()*. Este comando faz com que a tarefa que o executou, tenha seu processamento suspenso durante um certo tempo, voltando ao estado de pronta para executar quando é expirado tal intervalo de bloqueio.

O algoritmo e a implementação em linguagem C do Módulo de Escalonamento, relativo ao tratamento do comando *wait()*, são apresentados a seguir.

As tarefas que executam um comando *wait()* são inseridas numa Fila de Espera. Esta fila é organizada em ordem crescente, segundo o tempo de bloqueio das tarefas. Conforme transcorre a simulação, o tempo que resta para que as tarefas sejam

desbloqueadas, é atualizado. Quando o tempo restante atinge o valor 0, as respectivas tarefas são inseridas na Fila de Escalonamento, ficando disponíveis ao escalonador. O algoritmo detalhado é apresentado na sequência.

Algoritmo WAIT():

```
/* IMPORTANTE: Esta rotina é executada a cada "tick".                    */
Início
/* ------------------ Manutenção da Fila de Espera ---------------  */
  Se houver tarefas na Fila de Espera então
    Atualiza o tempo de bloqueio das tarefas;
    Enquanto há tarefa que já cumpriu o tempo de bloqueio faça
      Retire da Fila de Espera a tarefa com tempo de
      bloqueio expirado;
      Insira-a na Fila de Escalonamento adequada;
    FimEnquanto
  FimSe
          /* continua ...*/

/* ----------------- Tratamento do comando WAIT ------------------ */
  Se foi executado um WAIT() no instante atual da simulação então
   Lê o parâmetro Tempo de Bloqueio;
    /* Suspende o processamento da tarefa que executou WAIT */
    Retira a primeira tarefa da Fila de Pronto;
    Insere esta tarefa na Fila de Espera na ordem correta segundo
    seu Tempo de Bloqueio;
  FimSe
  Se há tarefas nas Filas de Escalonamento ou foi executado um comando WAIT() então
     Invocar o escalonador;
  Senão
    Não invocar o escalonador;
  FimSe
Fim
```

Na construção do Módulo de Escalonamento que trata do comando *wait()*, os recursos empregados serão:

1 - Filas Auxiliares;
2 - Campos Auxiliares;
3 - Comandos Genéricos.

A Fila de Espera é implementada através da Fila Auxiliar de número 0, que o SSE tem disponível. Associado a cada tarefa desta fila existe um campo chave para sua ordenação, que representa o tempo restante de bloqueio da tarefa. Para este campo, foi utilizado o Campo Auxiliar 0 do BCT da tarefa. Com o objetivo de otimizar a manutenção da Fila de Espera, esta foi implementada como "fila delta". Neste tipo de fila, as tarefas são organizadas em ordem crescente, sendo que o valor dos campos chaves (Tempo de Bloqueio) é acumulativo. Considere a figura 3.9 para ilustrar esta organização. Neste exemplo, existem três tarefas na Fila de Espera, 1,2 e 3, com os respectivos Tempo de Bloqueio, 5, 9 e 10 "ticks".

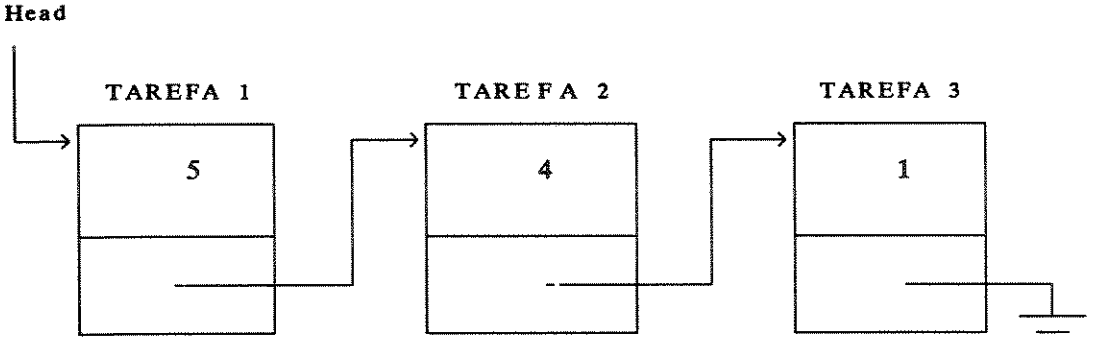



Figura 3.9 - Esquema da Fila de Espera

O valor do tempo de bloqueio de cada tarefa é calculado, somando-se o valor presente no campo chave da tarefa com os respectivos valores das tarefas que encontram-se a sua frente na Fila de Espera. Este cálculo pode ser visto a seguir para o exemplo da figura 3.9 (notação: TB(t) = tempo de bloqueio da tarefa t):

TB(1) = 5

TB(2) = 5 + 4 = 9

TB(3) = 5 + 4 + 1 = 10

A vantagem de ser trabalhar com filas tipo delta é que a manutenção do campo chave da primeira tarefa da fila , reflete diretamente na atualização dos campos das tarefas restantes.

Para simular o comando *wait()*, foi convencionado, para este caso, associá-lo ao *comando genérico* do tipo 1. Assim, cada ocorrência deste tipo de *comando genérico* representa a execução de um *wait()*. Observa-se que o tipo do *comando genérico* pode ser qualquer valor, uma vez que a rotina do usuário é que fará o reconhecimento do comando e, a seguir, seu tratamento.

O Módulo de Escalonamento, considerando o comando *wait()*, é mostrado abaixo. Como o interesse está apenas na implementação deste comando, a função de escalonamento (*scheduler()*) foi suprimida, sendo deixadas apenas aquelas relacionadas com o evento.

```
void userinit(void)
{
  /* Indica que o controle de invocação do escalonador será feito
  ** pela rotina de interrupção do usuário (userinter()).
  */
  set_sched_control_userint();

  /* inicializa a Fila de Espera                                    */
  /* nenhuma tarefa na fila                                         */
  set_aux_head(0,NULL);
  set_aux_tail(0,NULL);
}


Boolean userinter(void)
{
  /* ---------------------------------------------------------------
```

```
** Descrição: esta rotina é invocada a  cada  "tick"  lógico.
**             Ela realiza a manutenção da Fila de Espera, inter-
**             preta a ocorrência de um comando WAIT e também faz
**             seu tratamento.
**             A Fila de Espera é implementada através da Fila
**             Auxiliar de número 0. O valor chave para ordenação
**             da fila, é o Tempo de Bloqueio , o qual é armaze-
**             nado no Campo Auxiliar 0 da tarefa.
**             O Tempo de Bloqueio da tarefa é lido através  do
**             parâmetro 1 especificado quando foi introduzido o
**             o comando genérico.
**
** Variáveis:
**             WTask = tarefa no início da Fila de Espera;
**             NextWTask =tarefa seguinte a 1ª tarefa da Fila de
**                        Espera;
**             HeadTask =  1ª tarefa de uma fila;
**             TailTask =  última tarefa de uma fila;
**             WTime = Tempo de Bloqueio da tarefa;
**             WaitFlag =  indica se ocorreu (1) ou não (0) um co-
**                        mando WAIT.
**
** Notação:
**             FEP = Fila de Escalonamento Periódico
**             FA  = Fila Auxiliar
**             FE  = Fila de Espera
** ------------------------------------------------------------ */

TaskPtr         WTask, NextWTask;
TaskPtr         HeadTask, TailTask;
unsigned long WTime;
unsigned char WaitFlag;

/*----------------- Manutenção da Fila de Espera -------------- */
```

```
/* Pega a primeira tarefa da Fila de Espera                    */
WTask = get_aux_head(0);
/* Se houver tarefa na Fila de Espera                          */
if (WTask != NULL)
  {
    /* Atualiza o Tempo de Bloqueio das tarefas                */
    WTime = get_aux2_var(WTask,0) - 1;
    set_aux2_var(WTask,0,WTime);
    /* Enquanto houver tarefa com Tempo de Bloqueio expirado   */
    while (WTime == 0)
      {
        /* ------- retira esta tarefa da Fila de Espera -------  */
        NextWTask = get_aux_next(0,WTask);
        set_aux_head(0,NextWTask);
        /* Se a tarefa retirada for periódica                  */
        if (get_type_task(WTask) == p)
          {
            /* Insere a tarefa na Fila de Escalon. Periódico   */
            /* pega 1ª tarefa da FEP                           */
            HeadTask = get_psch_head
            /* Se não houver tarefas na FEP                    */
            if (HeadTask == NULL)
              /* insere a tarefa em 1º                         */
              set_psch_head(WTask);
            else
              {
                /* insere a tarefa na última posição           */
                TailTask = get_psch_tail();
                set_psch_next(TailTask,WTask);
              }
            /* atualiza o ponteiro de final da fila            */
            set_psch_tail(WTask);
            /* última tarefa da fila aponta para nada!         */
            set_psch_next(WTask,NULL);
```

```
        }
      /* Se a tarefa retirada for aperiódica                    */
      if (get_type_task(WTask) == a)
        {
          /* Insere a tarefa na Fila de Escalon. Aperiódico     */
          /* pega 1ª tarefa da FEA                              */
          HeadTask = get_asch_head
          /* Se não houver tarefas na FEA                       */
          if (HeadTask == NULL)
            /* insere a tarefa em 1º                            */
            set_asch_head(WTask);
          else
            {
              /* insere a tarefa na última posição            */
              TailTask = get_asch_tail();
              set_asch_next(TailTask,WTask);
            }
          /* atualiza o ponteiro de final da fila             */
          set_asch_tail(WTask);
          /* última tarefa da fila aponta para nada!          */
          set_asch_next(WTask,NULL);
        }
      /* pega a próxima tarefa da Fila de Espera              */
      Wtask = NextWTask;
      /* Se ainda houver tarefa                               */
      if (WTask != NULL)
        /* lê seu Tempo de Bloqueio                           */
        WTime = get_aux2_var(WTask,0);
      else  /* se não há mais tarefas                         */
        /* sai do loop                                        */
        WTime = 1;
    }
  }
/* ------------------------------------------------------------ */
```

```
/* ---------------- Tratamento do comando WAIT ---------------- */
WaitFlag = 0;
/* Se ocorreu um comando genérico do tipo WAIT                  */
if (get_type_command() == 1)
  {
    /* lê o Tempo de Bloqueio da tarefa                         */
    WTime = get_parameter(1);
    /* suspende a execução da tarefa                            */
    HeadTask = get_ready_head();
    set_ready_head(get_ready_next(HeadTask));
    insert_wait_queue(HeadTask,WTime);
    /* sinaliza a ocorrência do comando WAIT                    */
    WaitFlag = 1;
  }
/* ------------------------------------------------------------ */

/* Se houver tarefas nas Filas de Escalonamento ou              */
if ((get_psch_head() != NULL) || (get_asch_head() != NULL) ||
/*    tiver ocorrido um comando WAIT                            */
    (WaitFlag == 1))
  /*  invocar o escalonador                                     */
  return(true);
else       /*  do contrário                                    */
  /*  nada feito!                                              */
  return(false);
}

void insert_wait_queue(TaskPtr WTask, unsigned long WaitTime)
/*------------------------------------------------------------- */
/* Descrição: esta função insere uma dada tarefa na Fila de Espe- */
/*            considerando seu Tempo de Bloqueio.               */
/* Parâmetros: WTask = ponteiro para a tarefa a ser inserida;   */
/*             WaitTime = Tempo de Bloqueio da tarefa.          */
/*------------------------------------------------------------- */
```

```
{
  TaskPtr tptr1, tptr2, head;

  /* inicializa o campo chave (Tempo de Bloqueio) da tarefa          */
  set_aux2_var(0,WTask,WaitTime);
  /* pega a 1ª tarefa da Fila de Espera                              */
  head = get_aux_head(0);
  /* Se a Fila estiver vazia                                         */
  if (head == NULL)
    {
      /* inicializa a fila com a tarefa                              */
      set_aux_head(0,WTask);
      set_aux_next(0,WTask,NULL);
    }
  /* se há tarefas na Fila de Espera                                 */
  else
    {
      /* Se o Tempo de Bloqueio for menor que o da 1ª tarefa
                                                          da FE       */
      if (WaitTime < get_aux2_var(0,head))
        {
          /* insere a tarefa em 1º da FE                             */
          set_aux2_var(0,head,get_aux2_var(head) - WaitTime);
          set_aux_next(0,WTask,head);
          set_aux_head(0,WTask);
        }
      else
        {
          /* encontra a posição adequada para inserir a tarefa       */
          tptr2 = head;
          while ((get_aux_next(0,tptr2) != NULL) &&
                 ((get_aux2_var(0,tptr2) <= WaitTime) ||
                  (get_aux2_var(0,tptr2) == 0)))
            {
```

```
            tptr1 = tptr2;
            /* atualiza o campo chave conforme na estrutura de     */
           /* fila delta.                                          */
            WaitTime -= get_aux2_var(0,tptr2);
            set_aux2_var(0,WTask,WaitTime);
            tptr2 = get_aux_next(0,tptr2);
          }
        if ((get_aux_next(0,tptr2) == NULL) &&
            ((get_aux2_var(0,tptr2) <= WaitTime) ||
             (get_aux2_var(0,tptr2) == 0)))
          {
            /* insere a tarefa em último lugar                     */
            set_aux_next(0,tptr2,WTask);
            WaitTime -= get_aux2_var(tptr2);
            set_aux2_var(0,WTask,WaitTime);
            set_aux_next(0,WTask,NULL);
          }
        else
          {
            /* insere a tarefa numa posição no meio da FE          */
            set_aux_next(0,tptr1,WTask);
            set_aux_next(0,WTask,tptr2);
            set_aux2_var(0,tptr2,get_aux2_var(0,tptr2) - WaitTime);
          }
      }
  }
}
```

Os recursos flexíveis que o SSE dispõe, são suficientes para a simulação de uma grande variedade de eventos (comandos, funções, etc.) que provocam escalonamento, úteis às aplicações dos usuários.

### 3.3.7 - Considerações para Implementação do Módulo de Escalonamento

Com base nos dados apresentados para utilização do SSE como ambiente de implementação de algoritmos de escalonamento, pode-se observar uma série de considerações que são comuns à construção de qualquer Módulo de Escalonamento.

1 - O primeiro passo da implementação é determinar o tipo de controle de invocação do escalonador (via SSE ou rotina de interrupção do usuário *userinter()*). Se a função de escalonamento (*scheduler()*) necessita ser chamada apenas nos momentos em que há tarefas nas Filas de Escalonamento, deve-se optar que o controle de invocação desta função seja feito pelo SSE. Apesar de ser perfeitamente possível fazer este controle através da rotina de interrupção do usuário (*userinter()*), é preferível deixar este trabalho por conta do SSE, sempre que a situação permitir, o que torna a implementação do Módulo de Escalonamento mais simples.

2 - Independente da complexidade do Módulo de Escalonamento, este deve conter no mínimo as três funções já citadas: *userinit()*, *userinter()* e *scheduler()*.

3 - Respeitar a interface SSE - Módulo de Escalonamento. Qualquer acesso às estruturas de dados do SSE deve ser feito através dos serviços disponíveis.

4 - Observar a tipificação dos dados pré-definida do SSE (Ver apêndice A).

## 3.4 - Conclusão:

Neste capítulo foi apresentado o Sistema de Simulação para Escalonadores, enfocando os dois tipos de aplicação a que está destinado. Em um nível mais alto, o SSE é empregado como ferramenta de software que auxilia a análise de escalonamento de tarefas em sistemas de tempo real crítico. No segundo tipo de utilização, a nível de software básico, o SSE revela-se um ambiente interessante para a implementação de novos algoritmos de escalonamento e posterior análise. Na sequência, serão vistos os aspectos relevantes da implementação do SSE.

# Capítulo IV

# Implementação do SSE

# Capítulo IV

# Implementação do SSE

## 4.1 - Introdução

O SSE foi desenvolvido em uma máquina compatível ao IBM-PC trabalhando sob ambiente MS-DOS. A escolha desta linha de computadores foi devido a sua grande disseminação nos meios industriais, assim como nos centros de pesquisa. Como ferramenta de implementação, optou-se pelo uso da linguagem C a qual, em geral, é familiar às pessoas que trabalham com software básico. A portabilidade da linguagem C também foi outro fator que ponderou em sua escolha. Futuras migrações do SSE para outros ambientes não deverão apresentar maiores dificuldades.

Neste capítulo será discutida a estrutura funcional do SSE, apresentando os principais módulos que o compõem. Algumas estruturas de dados internas ainda não vistas serão descritas, considerando seu papel dentro do sistema.

## 4.2 - Estrutura Funcional do SSE

O SSE é composto por quatro módulos principais:

1 - Interface do Usuário
2 - Simulador
3 - Escalonadores Internos
4 - Interface SSE - Módulo de Escalonamento

O diagrama global de fluxo de dados do SSE e seu relacionamento externo pode ser visto na figura 4.1.

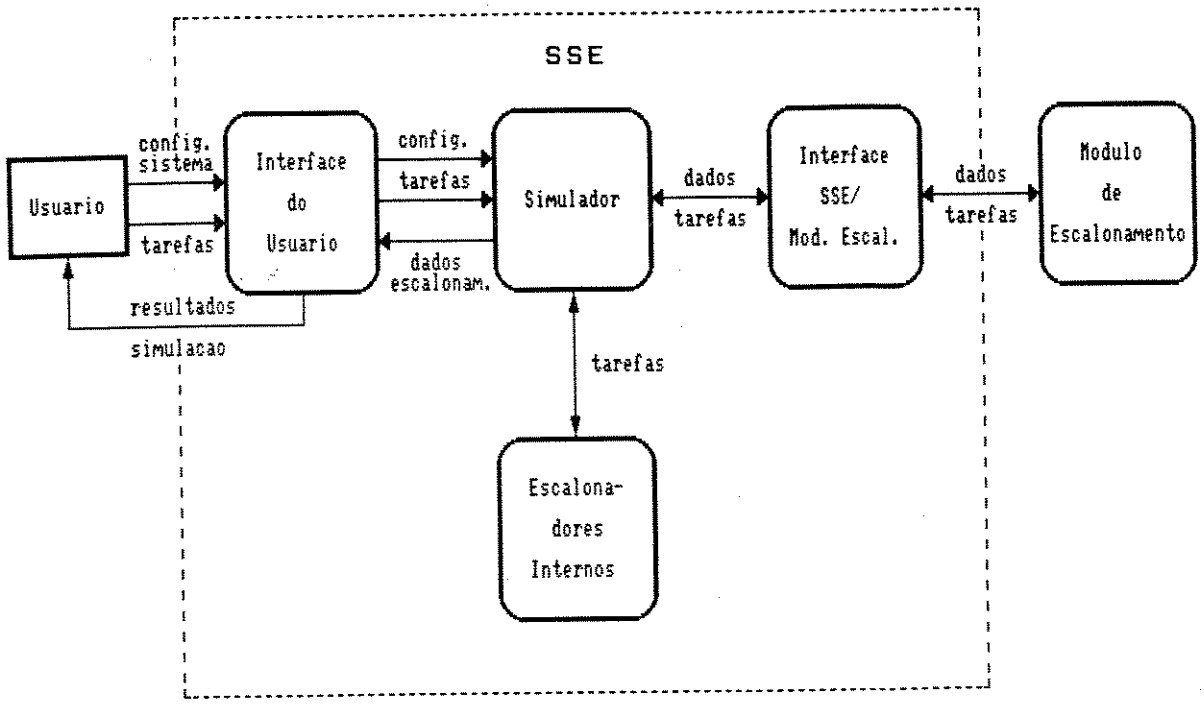



Figura 4.1 - Diagrama geral do SSE

A seguir serão descritos os aspectos mais relevantes de cada módulo do SSE.

### 4.2.1 - Interface do Usuário

A Interface do Usuário é responsável pela troca de informações entre o usuário e o SSE, quando se está em uma seção de simulação. Através dela, é realizada a edição de tarefas, configuração do sistema, análise de escalonamento e todos os demais serviços descritos no capítulo III. As informações de configuração do sistema, bem como os parâmetros das tarefas, são passados para o módulo Simulador. Este, por sua vez, retorna à Interface do Usuário os dados de escalonamento que são apresentados, numa forma adequada, como resultados da simulação.

Para facilitar o uso do SSE, a interação entre o usuário e o sistema é toda feita através de "menus" e "janelas", conceitos aplicados atualmente na maioria dos

softwares "amigáveis".

A construção deste módulo foi feita através de uma biblioteca de funções desenvolvidas especialmente para o SSE.

A ênfase dada à implementação da Interface do Usuário é justificada por sua importância dentro do sistema. Cada vez mais projetistas de software se preocupam com este aspecto. Independente da complexidade do sistema, se a interface for "pobre", sua operação será pouco estimulante, podendo levá-lo ao desuso.

### 4.2.2 - Escalonadores Internos

Neste módulo, encontram-se implementadas as políticas de escalonamento que o SSE tem disponível. Antes de iniciar a simulação, o usuário deve selecionar uma delas utilizando para isto a opção de escolha de escalonador, encontrada no menu de configuração do sistema.

O escalonador escolhido interage com o Simulador, mantendo a Fila de Pronto em ordem segundo um dado critério.

Os escalonadores que compõem este módulo são:

- *Earliest Deadline*
- *Least Laxity*
- *Taxa Monotônica*
- *Deferrable Server*
- *Prioridade Fixa Simples*
- *FIFO*

O código fonte destas políticas encontra-se no Apêndice B, e deve servir de orientação aos usuários do SSE na implementação de outras políticas de escalonamento.

### 4.2.3 - Interface SSE - Módulo de Escalonamento

Quando se está trabalhando com o SSE como ambiente para implementação de políticas de escalonamento, há a necessidade do escalonador em desenvolvimento inte-

ragir com o sistema a nível de suas estruturas de dados. Foi definida uma interface entre o SSE e o Módulo de Escalonamento através de um conjunto de serviços disponíveis ao usuário. O objetivo destes serviços é fornecer um acesso fácil e seguro às estruturas de dados do SSE que estão relacionadas com o escalonamento das tarefas dentro da simulação, sem que seja preciso conhecer detalhes da implementação do sistema. A definição destes serviços encontra-se na seção 3.3.4.

### 4.2.4 - Simulador

Este módulo, em conjunto ao Módulo de Escalonamento, constituem a base do SSE, simulando o funcionamento de um núcleo de tempo real. Suas principais funções se resumem no gerenciamento das tarefas e manipulação dos eventos do sistema.

Durante a execução, o Simulador interage com o Escalonador (interno ou externo) que, por sua vez, determina a ordem de processamento das tarefas (figura 4.1), e os resultados da simulação são passados à Interface do Usuário para serem exibidos.

O SSE é dirigido por um conjunto de eventos, citados abaixo:

1 - Invocação de Tarefa (Periódica ou Aperiódica)
2 - Fim de Execução de Tarefa
3 - Requisição de Rendezvous
4 - Resposta à Rendezvous
5 - Violação de Deadline
6 - Breakpoint
7 - Comando Genérico
8 - Evento do Usuário
9 - Fim da Simulação

Os eventos são gerados pelo Simulador, que também faz o tratamento dos mesmos através de ações particulares. Este procedimento, esquematizado na figura 4.2, é discutido a seguir.

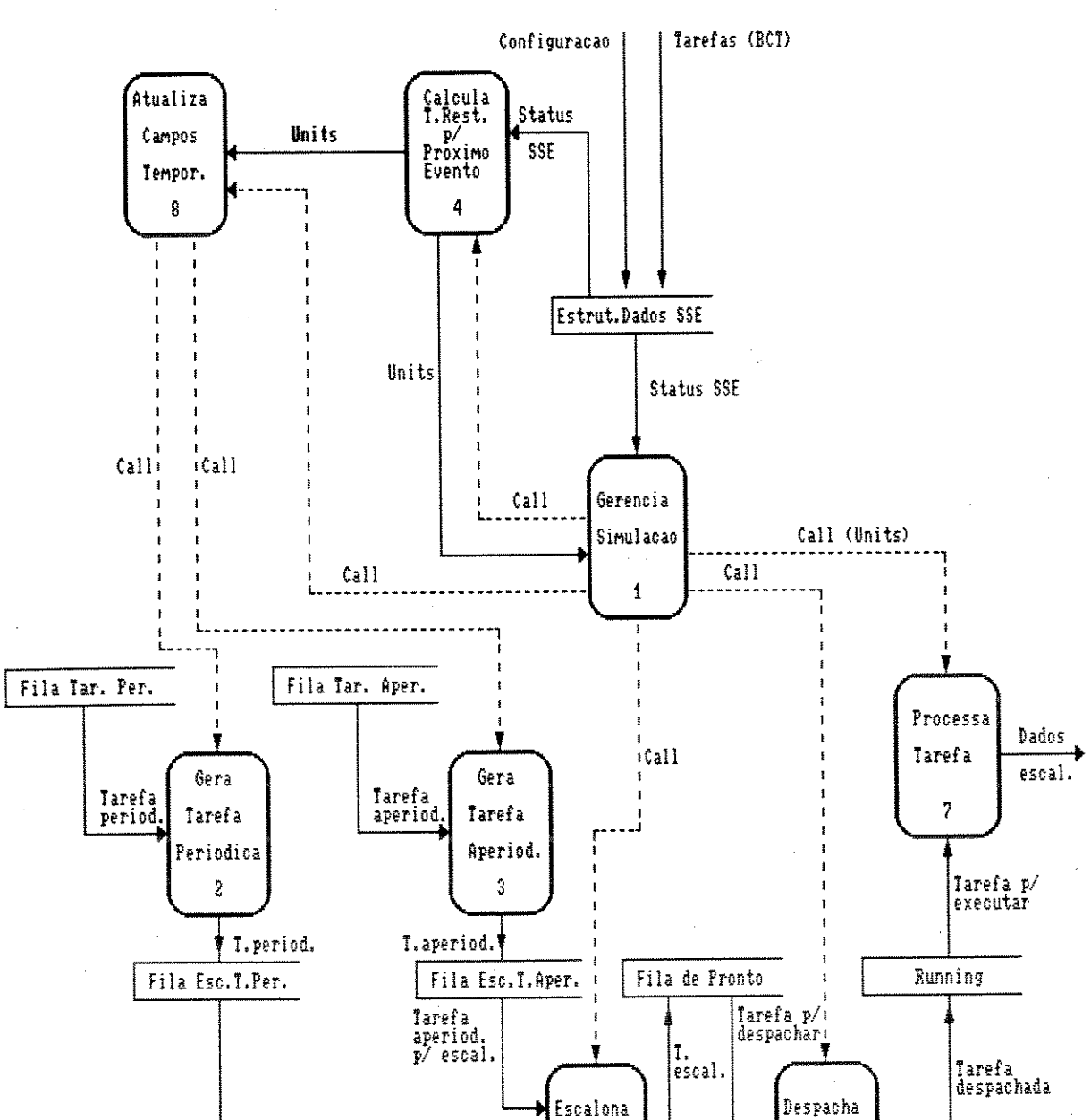



**Figura 4.2 – Diagrama de fluxo de dados do módulo Simulador**

#### 4.2.4.1 - Geração de Tarefas

As tarefas que fazem parte da simulação encontram-se inseridas na Fila de Tarefas Periódicas e na Fila de Tarefas Aperiódicas, conforme sua natureza. Estas filas são utilizadas para simular a chegada (invocação) de tarefas no sistema, e não devem ser confundidas com as Filas de Escalonamento Periódico e Aperiódico.

A Fila de Tarefas Periódicas é organizada em ordem crescente de período. Para maior eficiência, esta foi implementada como fila "delta", ou seja, o campo chave (período) é acumulativo. Como exemplo, considere uma simulação com quatro tarefas, T1, T2, T3 e T4, com os respectivos períodos: 50, 80, 230, 300 "ticks". A Fila de Tarefas Periódicas exibindo o campo chave de cada tarefa, é mostrada na figura 4.3.

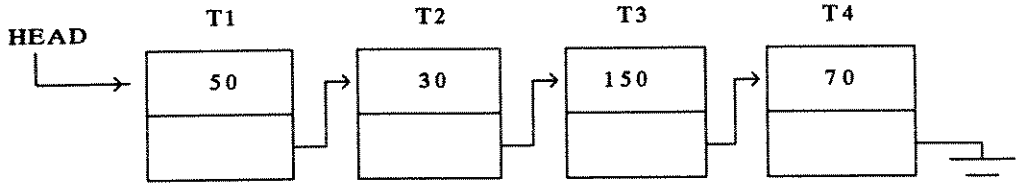
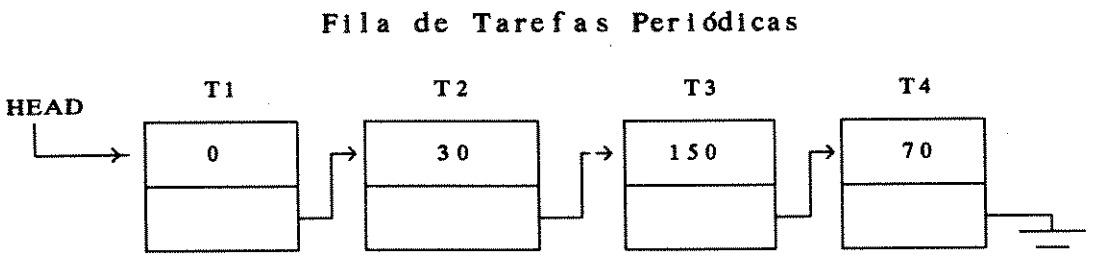



Figura 4.3 - Fila de Tarefas Periódicas

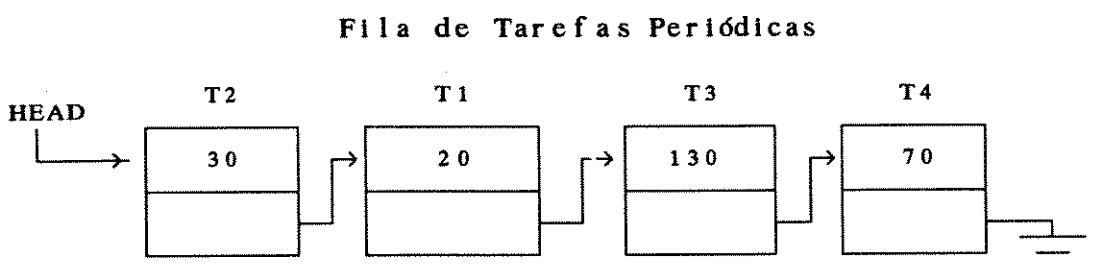
Conforme a simulação transcorre, o tempo é subtraido do campo chave da primeira tarefa da fila. Quando este atinge o valor zero, significa que a tarefa foi invocada (figura 4.4 (a)). O evento então é sinalizado ao processo Gerador de Tarefas Periódicas (2) que, por sua vez, atualiza a Fila de Tarefas Periódicas inserindo a tarefa novamente na fila, observando sempre a ordem desta e o conceito de fila "delta". A tarefa invocada também é colocada ao final da Fila de Escalonamento Periódico, ficando disponível ao Escalonador. Este processo é ilustrado na figura 4.4, para o mesmo conjunto de tarefas do exemplo anterior.

O uso de fila "delta" simplifica a manutenção de filas com campos temporizados, pois trabalha-se com o valor chave apenas da primeira tarefa, refletindo diretamente na atualização das demais. A dificuldade encontrada neste tipo de implementação é a operação de inserção que é um pouco mais elaborada que a inserção em filas comuns.

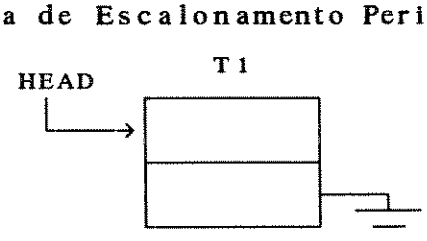
As tarefas periódicas que são inseridas no sistema possuem o mesmo tempo de pronto. Isto significa que, no início da simulação, todas as tarefas periódicas são invocadas simultaneamente. Esta opção de simulação não se trata de uma restrição, visto que a situação voltará a acontecer a cada múltiplo do m.m.c do período das tarefas. Quando for necessário que uma tarefa periódica inicie em um instante diferente do instante 0, pode-se utilizar o seguinte artifício: especifique um "Breakpoint" para o instante que se deseja que a tarefa períodica entre na simulação. Ao ser atingido o "Breakpoint", faça a inserção daquela tarefa periódica.




(a) - Tarefa T1 tem seu período vencido



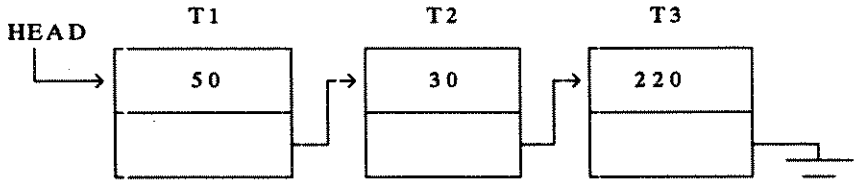

(b) - Tarefa T1 é reinserida na Fila de Tarefas Periódicas



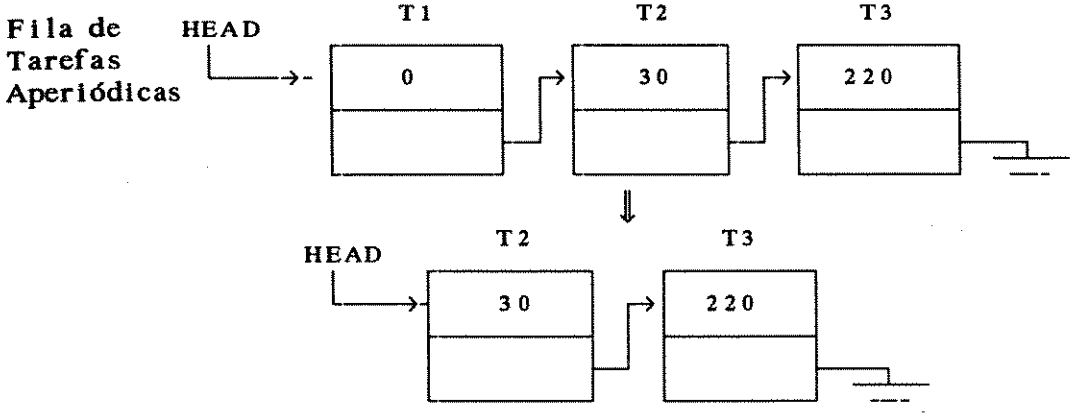

(c) - Tarefa T1 fica disponível ao Escalonador

Figura 4.4 - Geração de tarefas periódicas

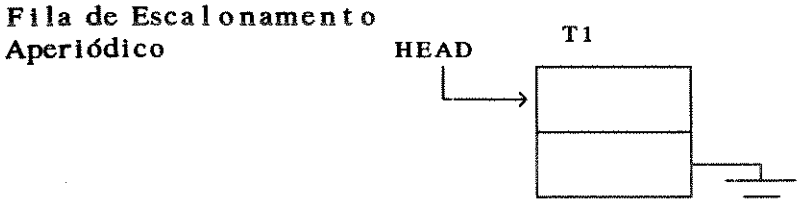
O processo de Geração de Tarefas Aperiódicas é semelhante ao descrito para tarefas periódicas. A Fila de Tarefas Aperiódicas é organizada em ordem crescente de instante de chegada e é implementada como fila "delta". Quando uma tarefa aperiódica é invocada, ela é retirada desta fila e colocada na Fila de Escalonamento Aperiódico. Esta tarefa não é mais inserida na Fila de Tarefas Aperiódicas. A figura 4.5 ilustra o processo para três tarefas, T1, T2 e T3 com os respectivos tempos de chegada 50, 80, 300.




(a) - Estado inicial da Fila de Tarefas Aperiódicas




(b) - Após 50 ticks, T1 é invocada e retirada da FTA




(c) - Tarefa T1 fica disponível ao Escalonador

Figura 4.5 - Geração de tarefas aperiódicas

#### 4.2.4.2 - Processo de Escalonamento

Quando há tarefas nas Filas de Escalonamento Periódico ou Aperiódico, o Escalonador pode ser ativado. O modo como isto será feito depende do controle de invocação selecionado (ver seção 3.3.2, capítulo III). Caso este controle seja executado

pelo SSE, a cada chegada de tarefa, o Escalonador é invocado. Se o controle for feito pelo Módulo de Escalonamento (fornecido pelo usuário), a invocação vai depender do(s) evento(s) escolhido(s) para isto, o que é particular da política implementada. Em geral, a chegada de tarefas é um evento significativo para se chamar o Escalonador (com exceção do caso de políticas de escalonamento não-preemptivas).

As tarefas escalonadas são inseridas na Fila de Pronto, indicando que se encontram aptas a serem executadas.

#### 4.2.4.3 - Despachador de Tarefas

Quando existe(m) tarefa(s) na Fila de Pronto, o processo Despachador é ativado pelo Gerenciador da Simulação. Sua função consiste em entregar ao módulo Processador as tarefas prontas, uma a uma. Para isto, o Despachador informa qual a primeira tarefa da Fila de Pronto. Observa-se que a tarefa informada não é retirada desta fila.

#### 4.2.4.4 - Execução das Tarefas

O módulo Processador (7) é responsável pela execução das tarefas. As tarefas são executadas até a ocorrência de algum dos eventos reconhecidos pelo SSE. Para isto, o Processador é informado pelo módulo Gerenciador, dentro de quantos "ticks" acontecerá um próximo evento. Por se tratar de simulação, a execução de uma tarefa é simbólica, e se traduz na atualização de um campo temporizado que informa quantos "ticks" faltam para o processamento da tarefa estar completo. Como resultado, são passados dados de escalonamento para a Interface do Usuário, que permitem o registro da execução das tarefas do sistema. Estes dados são:

- Instante Atual
- Tipo de Tarefa (Periódica ou Aperiódica)
- Identificador da Tarefa
- Até que Instante que a Tarefa Executou
- Deadline Absoluto

- Tempo Restante de Execução
- Evento Ocorrido

Um exemplo de relatório gerado é encontrado na seção 3.2.3 do capítulo III.

Dois eventos em particular são tratados pelo módulo Processador: requisição de rendezvous e resposta à rendezvous (apenas para o caso de tarefas que se comunicam por esse meio). Apesar de não haver diferença, a especificação de rendezvous como sendo de requisição ou resposta diz respeito apenas ao ato de iniciar e completar a comunicação, respectivamente.

Quando uma tarefa inicia um rendezvous (requisição), esta é retirada da Fila de Pronto, colocada em um estado de espera e o rendezvous é sinalizado ao seu correspondente na tarefa destino, como ilustrado na figura 4.6. Neste exemplo, a tarefa T1 executa um rendezvous com a tarefa T5 e como T1 é que inicia o processo, ela é suspensa e retirada da Fila de Pronto.

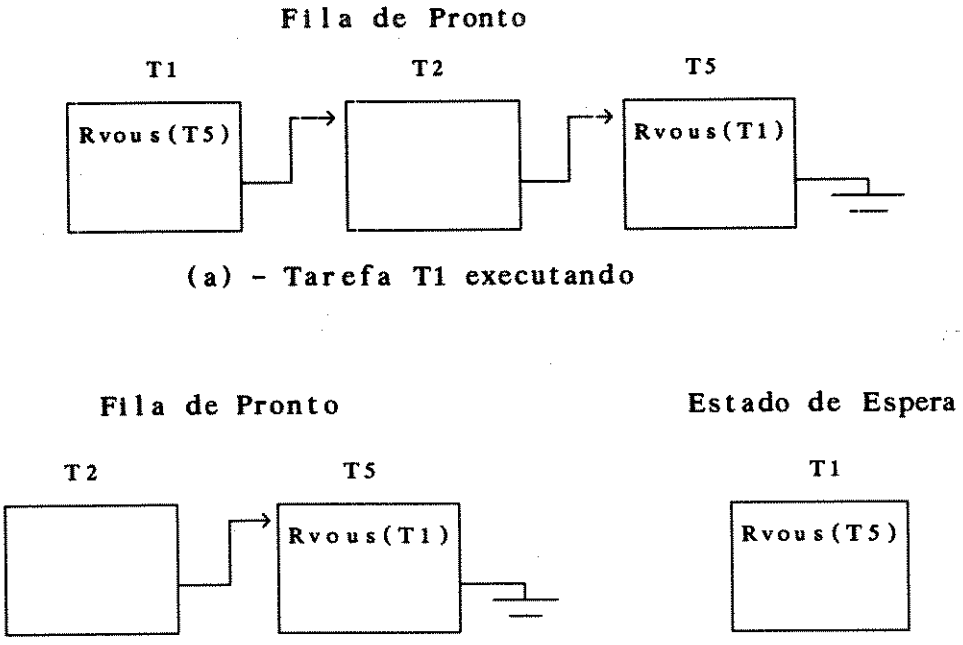
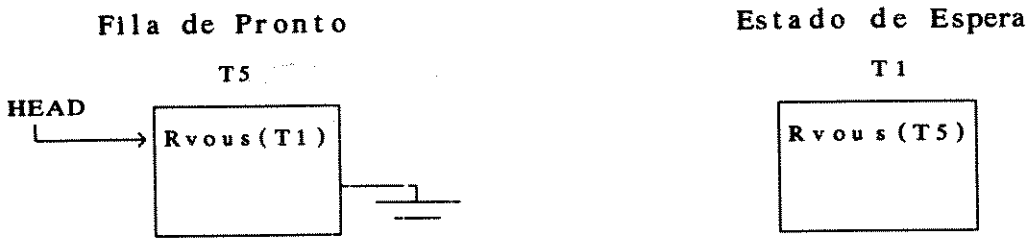
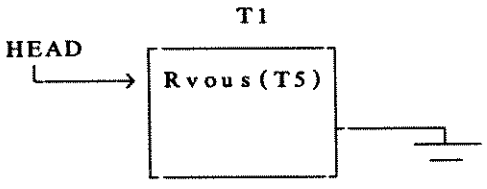



Figura 4.6 - Requisição de rendezvous

Quando uma tarefa executa um rendezvous completando a comunicação, a outra tarefa que aguardava resposta é inserida na Fila de Escalonamento correspondente, e a partir deste instante fica disponível ao Escalonador que, por sua vez, decidirá a sua inserção novamente na Fila de Pronto. Observa-se que a tarefa que finalizou o rendezvous, continua na Fila de Pronto, e será mantida em execução se esta tiver uma prioridade maior que a que aguardava a comunicação, conforme a política de escalonamento adotada. A figura 4.7 mostra o processo de resposta para rendezvous, no caso do exemplo dado na figura 4.6.




(a) - Tarefa T5 completa o rendezvous com T1




(b) - Tarefa T1 é inserida na Fila de Escalonamento

Figura 4.7 - Resposta a um rendezvous

#### 4.2.4.5 - Cálculo do Instante de Ocorrência de Eventos

A implementação dos eventos simulados no SSE é baseada em variáveis que indicam de alguma forma o tempo de ocorrência dos mesmos. Considerando os valores armazenados nestas variáveis, é calculado o número de "ticks" restantes (TR), a partir do instante atual, para ocorrer um próximo evento. Este dado é entregue ao Gerenciador da Simulação, que por sua vez, repassa o valor para o Processador para que este saiba quanto tempo que uma tarefa poderá ficar executando, antes que chegue um evento. A informação também é passada para o módulo que realiza a temporização do sistema.

#### 4.2.4.6 - Controle do Deadline das Tarefas

As tarefas que possuem prazo ("deadline") para terminar de executar, são acompanhadas por um processo que verifica se elas respeitam seus requisitos de tempo. Para isto, as tarefas prontas para executar são inseridas numa fila tipo "delta", denominada Fila de Deadlines, organizada por um campo que representa o tempo restante para que seja atingido o respectivo deadline. Conforme a simulação transcorre, este campo da primeira tarefa na fila é decrementado. Quando uma tarefa termina de ser processada, ela é retirada da fila e esta é atualizada. Como consequência, se o campo de deadline restante de uma tarefa chegar ao valor zero, significa que ela teve seu prazo vencido e não terminou de executar (pois se assim fosse, não estaria mais na fila). Nesta situação, a(s) tarefa(s) (pode haver mais de uma com o mesmo prazo) é (são) inserida(s) numa fila de tarefas com deadline violado, que pode ser consultada posteriormente, e a simulação é suspensa naquele instante.

O controle do deadline das tarefas é feito pelo módulo de Temporização do Sistema.

#### 4.2.4.7 - Temporização do Sistema

O processo de Temporização (8) feito no Simulador, consiste na atualização das variáveis relacionadas com os eventos, assim como na manutenção de algumas das filas de tarefas presentes no sistema. O valor contido nas variáveis temporizadas é contabilizado com a informação de número de "ticks" restantes (TR) para ocorrer um próximo evento (recebido do módulo 4), de forma que elas sempre reflitam a evolução do tempo para cada evento.

As variáveis atualizadas e seu respectivo papel dentro do SSE, são descritas a seguir:

*Clock*: é o relógio do sistema. Indica o número de "ticks" decorridos na simulação até o instante atual.

*RemSim*: é o tempo que resta para terminar a simulação.

*RemPeriod*: campo do BCT (Bloco de Controle da Tarefa) que representa o tempo que resta para o vencimento do período da tarefa.

*RemExec*: campo do BCT que indica o tempo que falta para a tarefa terminar de executar.

*RemDead*: campo do BCT que indica o tempo restante para vencer o prazo de execução (deadline) da tarefa (quando presente).

Em resumo, o módulo de Temporização do sistema, conhecendo o número de "ticks" compreendidos entre dois eventos consecutivos, atualiza as variáveis temporizadas com este valor, simulando assim o tempo transcorrido. Conforme o valor resultante da atualização, são tomadas ações específicas (por exemplo, gerar uma tarefa).

#### 4.2.4.8 - Gerenciamento da Simulação

O módulo de Gerenciamento (1) desempenha papel fundamental no decorrer de uma simulação. Ele é responsável pela coordenação dos módulos que compõem o Simulador, fazendo a invocação deles nos momentos adequados.

Como foi descrito na seção 3.3.2 o controle de invocação do escalonador, isto é, a decisão de chamá-lo ou não, pode ser feita pelo Módulo de Escalonamento (através da função de interrupção do usuário) ou pelo próprio SSE. De posse desta informação, o módulo de Gerenciamento controla a simulação de duas maneiras distintas. No primeiro caso, a execução da simulação é feita "tick a tick", ou seja, a atualização de tempo é feita a cada unidade de tempo transcorrido, e não através de uma única vez com o valor total de número de "ticks" entre dois eventos consecutivos. A cada "tick" a função de interrupção (*userinter()*) presente no Módulo de Escalonamento, é invocada. O processamento que esta função realiza é particular ao algoritmo de escalonamento. Em geral, são contabilizados campos temporizados relacionados com algum evento. A função de interrupção é responsável por informar ao Simulador se o escalonador deve ser invocado ou não, para isto ela retorna um valor booleano. Um ponto importante a observar é que, como a invocação do escalonador depende unicamente da decisão desta função, todo tipo de evento que cause um escalonamento, deve necessariamente ser considerado.

Caso o controle de invocação do escalonador seja feito pelo SSE, a cada ocorrência de um evento reconhecido, ele invoca o módulo Escalonador.

Se foi solicitada pelo usuário uma interrupção da execução da simulação, o módulo Gerenciador passa o controle para a Interface do Usuário, permitindo assim o acesso a qualquer dos serviços descritos no capítulo III (por exemplo, inserir uma nova tarefa na simulação). O módulo Gerenciador só terá de volta o controle da simulação, quando o usuário der o comando para *Continuar* ou *Iniciar* uma nova execução.

Seguindo este procedimento, o módulo Gerenciador controla a simulação até que atinja o prazo final especificado pelo usuário no instante da configuração do sistema.

# Capítulo V

# Conclusão

# CAPÍTULO V

# CONCLUSÃO

Os sistemas de tempo real crítico demandam por ferramentas que levem em consideração seus requisitos temporais. Neste sentido, o presente trabalho contribui com a ferramenta de software SSE, que é dedicada à área da teoria de escalonamento, a qual é de grande importância às aplicações de tempo real crítico.

O SSE possui duas finalidades principais:

1 - Ferramenta de análise de escalonamento de tarefas;
2 - Ambiente para implementação de políticas de escalonamento.

No primeiro caso, o SSE permite que seja analisado o comportamento das tarefas de uma aplicação, diante do uso de uma certa política de escalonamento, escolhida dentre as que o sistema oferece. Os recursos disponíveis encontrados no SSE asseguram uma grande flexibilidade à análise de escalonamento. A interface entre o sistema e o usuário, baseada em menus e janelas, torna a ferramenta muito fácil de operar o que, consequentemente, leva à obtenção de resultados mais rápidos.

Como ambiente para implementação de políticas de escalonamento, o SSE traz uma interface de software bem definida, que possibilita incorporar outros algoritmos de escalonamento, a fim de que sejam estudados, usando-se para isto as facilidades de análise que a ferramenta proporciona.

Os escalonadores *Taxa Monotônica*, *"Deferrable Server"* e *"Least Laxity"* foram implementados por um usuário que colaborou na fase de teste, validando as funções do SSE. Como experiência pode-se dizer que, tendo conhecimento completo da interface de software definida entre o SSE e o Módulo de Escalonamento, a implementação de escalonadores não traz grandes dificuldades.

Com relação à interface de software SSE - Módulo de Escalonamento, procurou-se fornecer os recursos mais requisitados à implementação de algoritmos de escalonamento. Os recursos flexíveis do SSE (Seção 3.3.6) fornecem suporte suficiente ao tratamento de eventos não previstos no sistema, assim como outros processamentos par-

ticulares ao algoritmo implementado.

Apesar do SSE ser dedicado à ambientes monoprocessadores, a possibilidade de se introduzir novos algoritmos de escalonamento, torna-o uma ferramenta de uso promissor nas pesquisas dentro da área da teoria de escalonamento aplicada à sistemas de tempo real crítico.

Quando foi projetado o SSE, foram incluidas as facilidades mais significativas que se julgou necessárias ao tipo de aplicação a qual foi destinado. A seguir são feitas algumas sugestões para a extensão do SSE que, certamente, o tornará mais interessente:

1 - Disponibilidade de outros mecanismos de sincronismo para as tarefas tais como: semáforos, monitores, primitivas p/ troca de mensagens, etc.; (atualmente estes mecanismos podem ser implementados através de *comandos genéricos* disponíveis no simulador);
2 - Incluir especificação para restrições de precedência entre tarefas; (atualmente pode ser simulado através do uso de rendezvous, presente no SSE);
3 - Incluir requisitos de recursos específicos às tarefas;
4 - Incluir novos escalonadores, envolvendo problemas de precedência e requisito de recursos;
5 - Geração de diagramas de execução (Gantt);
6 - Extensão do SSE para ambientes multiprocessadores e distribuídos;
7 - Migração do SSE para máquinas maiores (por exemplo, SUN Workstation). Graças a arquitetura modular em que foi construido o SSE, acredita-se que sua transposição para outros ambientes não deve apresentar grandes dificuldades.

# BIBLIOGRAFIA

# BIBLIOGRAFIA

[BEN 82] M. Ben-Ari, "Principles of Concurrent Programming", Prentice- Hall, Englewood Cliffs, N.J., 1982.

[CHE 87] S. Cheng, J. A. Stankovic, and K. Ramamritham, "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey", Real-Time Systems Newsletter, Vol. 3, No 2, Summer 1987, pp. 1 - 24.

[COE 86] J. M. A. Coello, "Suporte de Tempo Real para um Ambiente de Programação Concorrente", Dissertação de Mestrado, FEE - UNICAMP, 1986.

[FAU 88] S. R. Faulk, and D. L. Parnas, "On Synchronization in Hard-Real-Time Systems", Communications ACM, Vol. 31, March 1988.

[GAN 83] C. Gane, e T. Sarson, "Análise Estruturada de Sistemas", LTC, RJ, 1986.

[GEN 89] E. A. Gene, "COINS 520 Project", Technical Report, December 1989.

[HSI 91] Y. Z. Hsieh, "Estudo de Algoritmos de Escalonamento para Aplicações de Tempo Real Crítico", Dissertação de Mestrado, FEE - UNICAMP, 1991.

[LEH 87] J. P. Lehoczky, L. Sha, and J. K. Stronider, "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", IEEE Real-Time Systems Symposium, December 1987, pp. 261 - 270.

[LEH 89] J. P. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm - Exact Characterization and Average Case Behavior", Proc. IEEE Real-Time System Symposium, CS Press, Los Alamitos, California, Order No. 2004, 1989, pp. 166 - 171.

[LEU 82] J. Y.-T. Leung and J. Whitehead, On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks", Performance Evaluation 2, pp. 237 - 250.

[LIU 73] C. L. Liu, and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real Time Environment" J. ACM, Vol. 20, No. 1, 1973, pp. 46-61.

[MS1 87] "Microsoft C for the MS-DOS Operating System - Language Reference Version 5.1", Microsoft Corporation, 1987.

[MS2 87] "Microsoft C for the MS-DOS Operating System - Run-Time Library Reference Version 5.1", Microsoft Corporation, 1987.

[MS3 87] "Microsoft C Optmizing Compiler for the MS-DOS Operating Systems - User's Guide", Microsoft Corporation, 1987.

[MS4 87] "Microsoft CodeView and Utilities", Microsoft Corporation, 1987.

[MAG 86] M. F. Magalhães, "Software para Tempo Real", Editora da UNICAMP, Campinas, 1986.

[MOK 85] A. K. Mok, "SARTOR - a Design Environment for Real-Time Systems", Proc. 9th IEEE COMPSAC, october 1985, pp. 174 - 181.

[PRE 87] R. S. Pressman, "Software Engineering - A Practitioner's Approach", MacGraw-Hill Inc., 1987.

[STU 89] D. A. Stuart, "Implementing a Verifier fo Real-Time Systems", Technical Report, Department of Computer Sciences, University of Texas at Austin. Austin, Texas 78712.

[RAJ 88] R. Rajkumar, L. Sha, J. P. Lehoczky, and K. Ramamritham, "An Optimal Priority Inheritance Protocol for Real-Time Synchronization", Technical Report, pp. 1-24.

[RAM 84] K. Ramamritham, and J. A. Stankovic, "Dynamic Task Scheduling in Hard Real-Time Distributed Systems", IEEE Software, Vol. 1, July 1984, pp. 65 - 75.

[RAM 89] K. Ramamritham, J. A. Stankovic, "Overview of the SPRING Project", Technical Report, April 1989, pp. 18 - 45.

[SHA 87] L. Sha, R. Rajhumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach ro Real Time Synchronization", Technical Report, Department of Computer Science, CMU, 1987.

[SHA 89] L. Sha, R Rajkumar, J. P. Lehoczky, and K. Ramamritham, "Mode Change Protocols for Priority-Driven Preenptive Scheduling", COINS Technical Report 89-60, pp. 1 - 27.

[SHA 90] L. Sha, and J. B. Goodenough, "Real-Time Scheduling Theory and Ada", Computer, April 1990, pp. 53 - 62.

[SPR 89] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic Task Scheduling for Hard Real-Time Systems", J. Real-Time Systems, Vol. 1, No. 1, 1989, pp. 27 - 60.

[STA1 88] J. A. Stankovic, And W. Zhao, "On Real-Time Transactions", Sigmod Record, Vol. 17, No. 1, March 1988, pp. 4 - 17.

[STA2 88] J. A. Stankovic, "Misconceptions About Real-Time Computing", Computer, October 1988, pp. 10 - 19.

[TOK 88] H. Tokuda, and M. Kotera, "A Real-Time Tool Set for the ARTS Kernel", Technical Report.

[XU 90] J. Xu, and D. L. Parnas, "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations", IEEE Transactions on Software Engineering, Vol. 16, No 3, March 1990, pp. 360 - 369.

# APÊNDICES

# APÊNDICE A - ESPECIFICAÇÕES TÉCNICAS DO SSE

## A.1 - Informações gerais

Nome do sistema: SSE
Linha de computadores: compatíveis IBM-PC
Linguagem de desenvolvimento: C
Compilador: Microsoft C, versão 5.1
Modelo de memória usado: "Large"
Depurador: Microsoft CodeView
Tamanho do código executável: ~ 125 Kb
Tempo de desenvolvimento: 7 meses
Módulos que compõem o SSE:
SSE.C, SSE2.C, SSE3.C, SSE4.C, SSE5.C,
SSE6.C, SSE7.C, SSE8.C
Arquivos de include: SSE.H, SCREEN.H
Biblioteca utilizada: SCREEN.LIB

## A.2 - Descrição dos Módulos

SSE, SSE2, SSE3, SSE4, SSE5: interface com o usuário e execução da simulação.
SSE6: serviços da interface SSE - Módulo de Escalonamento.
SSE7: Módulo de Escalonamento definido pelo usuário.
SSE8: Escalonadores internos do SSE.
SSE.H: arquivo de include do SSE. Contém constantes e principais estruturas de dados utilizadas pelo sistema.
SCREEN.H: arquivo de include referente a biblioteca SCREEN.
SCREEN.LIB: biblioteca de funções utilizadas na interface do usuário.

O SSE7.C contém o Módulo de Escalonamento, assim quando se estiver implementando um algoritmo externo ao sistema, as novas rotinas devem substituir aquelas encontradas neste arquivo.

### A.3 - Como Compilar um Módulo

Para compilar um módulo usando o compilador Microsoft C basta dar o seguinte comando diretamente do DOS:

cl /AL /c MODULO.C

Por exemplo, para compilar o Módulo de Escalonamento seria:

cl /AL /c SSE7.C

### A.4 - Como "Linkar" os Módulos

Uma vez que todos os módulos da aplicação estejam compilados (.OBJ), para "linka-los" basta:

cl /AL SSE SSE2 SSE3 SSE4 SSE5 SSE6 SSE7 SSE8 SCREEN.LIB

Observa-se que existem outras maneiras de linkar uma aplicação, assim, fica a gosto do usuário a que achar mais conveniente, contanto que não se esqueça de nenhum dos módulos acima, e também da biblioteca SCREEN.LIB.

### A.5 - Limitações

A quantidade de tarefas que o SSE pode trabalhar depende apenas do espaço de memória disponível, uma vez que cada tarefa é representada por uma estrutura de dados (BCT). No arquivo de include SSE.H contém constantes que determinam o número máximo de tarefas periódicas e aperiódicas, que conforme a necessidade da aplicação, podem ser aumentadas ou diminuidas. A seguir é dada as constantes definidas , os respectivos significados e valores "default".

MAXPTASKS = nro máximo de tarefas periódicas (default = 50)
MAXATASKS = nro máximo de tarefas aperiódicas (default = 50)
MAXRV = nro máximo de rendezvous definidos em cada tarefa (default = 10)
MAXVARS = nro de campos auxiliares por tarefa (default = 5)
MAXQ = nro de filas auxiliares disponíveis ao usuário.

# APÊNDICE B - EXEMPLOS DE IMPLEMENTAÇÕES DE ESCALONADORES

Este apêndice contém as implementações dos algoritmos internos do SSE, seguindo a interface de software definida. Estes exemplos servem de base aos usuários que desejam implementar novas políticas de escalonamento. Foram omitidos o "Earliest Deadline" e o "Least Laxity" pois já foram apresentados no capítulo III.

**(A) - Taxa Monotônica**

```
#include <stdio.h>
#include "screen.h"
#include "sse.h"

void user_init(void)
{
  /* Indica que o controle de invocacao do escalonador sera feito
  ** pelo SSE                                                    */
  set_sched_control_SSE();
}


Boolean user_inter(void)
{
  /*
  ** Esta rotina não tem qualquer funcao para o Escalonador Taxa
  ** Monotonica, constando aqui apenas para compatibilidade com a
  ** interface definida
  */
  return(true);
}
```

```
void scheduler(void)
/* Politica de escalonamento: Taxa Monotonica                    */
{
  unsigned long SchPeriod;
  TaskPtr TPtr1,TPtr2,SHead,RHead;

  /* Escalonamento de tarefas periodicas                         */
  /* Pega a primeira tarefa da FPE                               */
  SHead = get_psch_head();
  /* Enquanto há tarefas para escalonar faça                     */
  while (SHead != NULL)
    {
      /* pega a primeira tarefa da FP                            */
      RHead = get_ready_head();
      /* Se não há tarefas na fila                               */
      if (RHead == NULL)
        {
          /* inicializa a FP com a tarefa                        */
          set_ready_head(SHead);
          set_ready_next(SHead,NULL);
        }
      else /* tem tarefas na FP                                  */
        {
          /* lê o período da tarefa a ser escalonada ...         */
          SchPeriod = get_period(SHead);
          /* Se o período dela for menor que o da 1ra tarefa da
             FP                                                  */
          if (SchPeriod < get_period(RHead))
            {
              /* insere em 1ro na FP                             */
              set_ready_next(SHead,RHead);
              set_ready_head(SHead);
            }
          else /* insere na devida posição na FP                 */
```

```
            {
              /* busca a posição onde inserir                    */
              TPtr2 = get_ready_head(); /* lê 1ra tarefa da FP   */
              /* Enquanto há tarefa na FP e o período da tarefa
              ** for maior ou igual a da tarefa lida na FP       */
              while ((TPtr2 != NULL) &&
                     (SchPeriod >= get_period(TPtr2)))
                {
                  /* passa para a seguinte tarefa da FP          */
                  TPtr1 = TPtr2;
                  TPtr2 = get_ready_next(TPtr2);
                }
              /* faz a inserção da tarefa na FP                  */
              set_ready_next(TPtr1,SHead);
              set_ready_next(SHead,TPtr2);
            }
        }
      /* pega a próxima tarefa a ser escalonada                  */
      SHead = get_psch_next(SHead);
      /* atualiza o ponteiro de inicio da FEP                    */
      set_psch_head(SHead);
    }
}
```

**(B) - "Deferrable Server"**

```
#include <stdio.h>
#include "screen.h"
#include "sse.h"

Boolean ApReady;
TaskPtr Server;

void user_init(void)
{
  TaskPtr Server;

  /* informa que o controle de invocação do escalonador será feito
  ** pelo Módulo de Escalonamento                                  */
  set_sched_control_userint();
  EventStep = 1;
  /* Assume que a tarefa Servidora será a de nro 1                 */
  Server = get_ptask_adrs(1); /* pega o endereço da tarefa 1       */
  /* e "seta" o tipo da tarefa como sendo s = especial             */
  set_task_type(Server,s);
  ApReady = false;
}

Boolean user_inter(void)
{
  unsigned long capacidade;
  TaskPtr TPtr1, TPtr2, OldRunning;

  /* pega a tarefa que estava executando no último instante        */
  OldRunning = get_last_running();
  /* Se a tarefa que executou é aperiódica                         */
  if ((OldRunning != NULL) &&
      (OldRunning->PerType == a))
```

```
  {
    /* atualiza a capacidade de serviço da tarefa servidora          */
    capacidade = get_cap_service(Server);
    capacidade--;
    set_cap_service(Server, capacidade);
    if (OldRunning != get_ready_head())
      ApReady = false;
  }
/* Se venceu o período da tarefa servidora                          */
  if (get_time() % get_period(Server) == 0)
  {
    /* Retirar tarefa servidora da Fila de Escal. Periódico         */
    TPtr2 = get_psch_head();
    if (TPtr2 != NULL)
    {
      while (TPtr2->PerType != s)
      {
        TPtr1 = TPtr2;
        TPtr2 = get_psch_next(TPtr2);
      }
      if (TPtr2 == get_psch_head())
        set_psch_head(get_psch_next(TPtr2));
      else
        set_psch_next(TPtr1, get_psch_next(TPtr2));
    }
  }
  /* ------------------------------------------------------------- */
  /* Se a capacidade de serviço da tarefa servidora se esgotou e
  ** a tarefa que estava executando era aperiódica e esta não
  ** terminou de executar então                                    */
  if ((get_cap_service(Server) == 0) &&
      (OldRunning != NULL) &&
      (OldRunning->PerType == a) &&
      (get_rem_exec(OldRunning) > 0))
```

```
{
  /* Retira a tarefa aperiodica da Fila de Pronto e a coloca
  ** na Fila de Escalonamento Aperiódico                          */
  ApReady = false;
  /* atualiza a Fila de Pronto                                    */
  set_ready_head(get_ready_next(OldRunning));
  /* Atualiza a Fila de Escalonamento Aperiódico                  */
  TPtr1 = get_asch_head();
  if (TPtr1 == NULL)
  {
    set_asch_head(OldRunning);
    set_asch_next(OldRunning, NULL);
  }
  else
  {
    set_asch_next(OldRunning, TPtr1);
    set_asch_head(OldRunning);
  }
  /* informa ao Escalonador para realizar novo escalonamento      */
  return(true);
}
/* lê a 1ra tarefa na Fila de Escal. Aperiódico                   */
TPtr1 = get_asch_head();
/* Se há tarefas na FEA e a tarefa servidora tem capacidade de
** serviço                                                        */
if ((TPtr1 != NULL) &&
    (!ApReady) &&
    (get_cap_service(Server) > 0))
{
  ApReady = true;
  /* Atualiza a Fila de Escalonamento Aperiódico                  */
  set_asch_head(get_asch_next(TPtr1));
  /*
  ** tarefa aperiódica fica com a mesma prioridade da tarefa
```

```
    ** servidora; artifício: faz-se  o  campo  período da tarefa
    ** aperiódica ter o mesmo valor que o da tarefa servidora.
    */
    set_period(TPtr1, get_period(Server));
    /* Insere a tarefa aperiódica na Fila de Escalon. Periódico      */
    TPtr2 = get_psch_head();
    if (TPtr2 == NULL)
    {
      set_psch_head(TPtr1);
      set_psch_next(TPtr1, NULL);
    }
    else
    {
      set_psch_next(TPtr1, TPtr2);
      set_psch_head(TPtr1);
    }
    /* ------------------------------------------------------------ */
  }
  /* Se houver tarefas na Fila de Escalonamento Periódico           */
  if (get_psch_head() != NULL)
    /* invoca o escalonador                                         */
    return(true);
  else
    /* não invoca o escalonador                                     */
    return(false);
}

void scheduler (void)
/* Política de escalonamento: Taxa Monotônica                       */
{
  /* esta rotina é a mesma empregada no Módulo de Escalonamento
  ** Taxa Monotônica                                               */
}
```

**(C) - Prioridade Fixa Simples**

```
#include <stdio.h>
#include "screen.h"
#include "sse.h"

void user_init(void)
{
  /* Indica que o controle de invocação do escalonador será feito
  ** pelo SSE                                                    */
  set_sched_control_SSE();
}

Boolean user_inter(void)
{
  /*
  ** Esta rotina não tem qualquer funcao para o Escalonador Prio-
  ** ridade Fixa Simples, constando aqui apenas para compatibili-
  ** dade com a interface definida
  */
  return(true);
}

void scheduler(void)
/* Politica de escalonamento: Prioridade Fixa Simples.          */
{
  unsigned char prior_sched; /* prioridade da tarefa escalonada  */
  TaskPtr TPtr1, TPtr2,  /* ptros p/ tarefas auxiliares          */
          SHead,  /* tarefa que está sendo escalonada            */
          RHead ; /* 1ra tarefa da Fila de Pronto (FP)           */
  int queue;  /* Fila de Escalonamento que se está trabalhando:  */
              /*          1 = FEP         2 = FEA                */
```

```
/* Para as Filas de Escalonamento Periódico e Aperiódico          */
for (queue = 1; queue <= 2; queue++)
  {
    if (queue == 1)  /* FEP                                       */
      /* pega a tarefa na FEP para escalonar                      */
      SHead = get_psch_head();
    else             /* FEA                                       */
      /* pega a tarefa na FEA para escalonar                      */
      SHead = get_asch_head();
    /* Enquanto há tarefas p/ escalonar faça                      */
    while (SHead != NULL)
      {
        /* lê a 1ra tarefa da FP                                  */
        RHead = get_ready_head();
        /* Se a fila não possui tarefas                           */
        if (RHead == NULL)
          {
            /* inicialize a FP com a tarefa escalonada          */
            set_ready_head(SHead);
            set_ready_next(SHead,NULL);
          }
        else /* FP já tem tarefas                                 */
          {
            /* insere tarefa na posição adequada dentro da FP   */
            /* lê a prioridade da tarefa escalonada             */
            prior_sched = get_priority(SHead);
            /* Se a prioridade da tarefa escal. for menor que
            **     a da 1ra tarefa da FP                        */
            if (prior_sched < get_priority(RHead))
              {
                /* insere a tarefa em 1ro na FP                 */
                set_ready_next(SHead,RHead);
                set_ready_head(SHead);
              }
```

```
            else /* senão                                          */
              {
                /* encontra a posição adequada e ...              */
                TPtr2 = get_ready_head();
                while ((TPtr2 != NULL) &&
                       (prior_sched >= get_priority(TPtr2)))
                  {
                    TPtr1 = TPtr2;
                    TPtr2 = get_ready_next(TPtr2);
                  }
                /* insere a tarefa na FP                          */
                set_ready_next(TPtr1,SHead);
                set_ready_next(SHead,TPtr2);
              }
          }

        /* Se a fila atual é a FEP                                */
        if (queue == 1)
          {
            /* pega a próxima tarefa na FEP p/ escanolonar        */
            SHead = get_psch_next(SHead);
            set_psch_head(SHead);
          }
        else
          {
            /* pega a próxima tarefa na FEA p/ escanolonar        */
            SHead = get_asch_next(SHead);
            set_asch_head(SHead);
          }
      }
  }
}
```

**(D) - FIFO**

```
#include <stdio.h>
#include "screen.h"
#include "sse.h"

void user_init(void)
{
  /* Indica que o controle de invocação do escalonador será feito pelo SSE */
  set_sched_control_SSE();
}

Boolean user_inter(void)
{
  /*
  ** Esta rotina não tem qualquer funcao para o Escalonador FIFO
  ** constando aqui apenas para compatibilidade com a interface definida
  */
  return(true);
}

void scheduler(void)
/* Política de escalonamento: FIFO                                    */
{
  unsigned char prior_sched;
  TaskPtr TPtr1, SHead, RHead;
  int queue; /* indica a Fila de Escalonamento: 1 = FEP,2 = FEA    */

  for (queue = 1; queue <= 2; queue++)
    {
      if (queue == 1)  /* Fila de Escal. Periodico                 */
        /* pega 1ra tarefa p/ escalonar                            */
        SHead = get_psch_head();
      else             /* Fila de Escal. Aperiodico                */
```

```
    /* pega 1ra tarefa p/ escalonar                    */
    SHead = get_asch_head();
  /* Enquanto há tarefas p/ escalonar                  */
  while (SHead != NULL)
    {
      /* lê a 1ra tarefa da Fila de Pronto             */
      RHead = get_ready_head();
      /* Se não houverem tarefas na FP                 */
      if (RHead == NULL)
        /* inicializa a FP com a tarefa escalonada     */
        set_ready_head(SHead);
      else
        {
          /* insere no final da FP                     */
          TPtr1 = RHead;
          while(get_ready_next(TPtr1) != NULL)
            TPtr1 = get_ready_next(TPtr1);
          set_ready_next(TPtr1,SHead);
        }
      set_ready_next(SHead,NULL);
      /* Pega próxima tarefa para escalonar            */
      if (queue == 1)
        {
          SHead = get_psch_next(SHead);
          set_psch_head(SHead);
        }
      else
        {
          SHead = get_asch_next(SHead);
          set_asch_head(SHead);
        }
    }
  }
}
```