UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

UMA ESTRATÉGIA DE ESCALONAMENTO DE PROCESSOS PERIÓDICOS E ESPORÁDICOS EM SISTEMAS DE TEMPO REAL CRÍTICO MONOPROCESSADOS

Autor: Bel. Alencar de Melo Júnior

Orientador: Prof. Dr. Maurício Ferreira Magalhães

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

- Janeiro de 1993 -

A minha família

AGRADECIMENTOS

Ao Maurício pela orientação, incentivo e amizade que sempre me dispensou.

A todos os professores com quem convivi e ao povo de meu país, responsáveis pela minha formação, toda ela realizada em escolas públicas.

A Cláudio Canhette e Paulo Minoru pela amizade, convivência e apoio, desde a época de graduação em São Carlos.

Aos amigos do grupo de tempo real, Rodrigo Spanó, Hsieh, Sibelius, Rosa, Juan e Adílson pela convivência e importantes sugestões e discussões.

Aos muitos amigos que fiz na FEE/UNICAMP, fundamentais para restituir o ânimo que as vezes teimava em esvair-se.

A todos os funcionários da FEE/UNICAMP, sempre atenciosos e prontos a colaborar.

ÍNDICE

RESUMO

Em sistemas de tempo real crítico os processos a serem escalonados estão sujeitos a um grande número de restrições: tempo de pronto, "deadline", relações de precedência e relações de exclusão mútua. O problema de escalonar um conjunto de processos em um sistema monoprocessador sujeito a estas restrições é conhecido ser "NP-hard", o que efetivamente impede o escalonamento destes em modo totalmente "on-line". Para os processos periódicos, utiliza-se um algoritmo já existente, projetado para ser usado por um escalonador "off-line", que resolve o problema citado anteriormente. Os processos esporádicos possuem tempo de pronto não determinístico, e desta forma, não podem ser escalonados "off-line", fazendo-se necessário uma abordagem "on-line". Este trabalho mostra que a implementação do escalonador "off-line" é factível e complementa este, propondo um procedimento eficiente para o atendimento "on-line" de processos esporádicos de modo a não comprometer o escalonamento gerado em modo "off-line" para os processos periódicos.

CAPÍTULO 1

I N T R O D U Ç Ã O

INTRODUÇÃO

Existem diversas aplicações de computadores nas quais as computações devem satisfazer rígidas restrições temporais, ou seja, deve ser garantido que estas computações sejam completadas antes de prazos especificados. Falhas em satisfazer os prazos especificados podem provocar uma degradação do sistema que, em algumas aplicações, ocasionam perda de vidas humanas ou grandes prejuízos financeiros. Sistemas nos quais a sua correção não depende apenas dos resultados lógicos das computações, mas também dos instantes nos quais estes resultados são produzidos, devendo cumprir prazos especificados, são chamados *Sistemas de Tempo Real Crítico* - **STRC**.

Sistemas de Tempo Real Crítico têm uma variada gama de aplicações que vão desde, por exemplo, complexos sistemas distribuídos responsáveis pelo tráfego aéreo de um país inteiro, até sistemas mais simples, como os presentes em motores de automóveis, eletroportáteis, etc., que vão se tornando cada vez mais presentes no dia a dia das pessoas graças à popularização da tecnologia de microprocessadores.

Procura-se obter a correção lógica de **STRC** através de técnicas adequadas de especificação, projeto e verificação, enquanto a teoria de escalonamento busca atender o cumprimento dos requisitos temporais especificados. Liu e Layland[01] apresentaram dois algoritmos clássicos de escalonamento para **STRC** monoprocessados: *Taxa Monotônica* e *"Earliest Deadline"*. Estes algoritmos são comprovadamente ótimos[01], no sentido que sempre que existir algum escalonamento possível para um conjunto de processos, estes algoritmos encontram uma solução. Contudo, estes algoritmos são ótimos apenas em cenários bastante restritos, difíceis de serem encontrados em aplicações reais.

Em **STRC** com alguma complexidade os processos a serem

escalonados estão sujeitos a um grande número de restrições: tempo de pronto(instante mais cedo no qual o processo torna-se disponível para execução), "deadline"(prazo máximo para término de execução de um dado processo), relações de precedência e relações de exclusão mútua. O problema de escalonar um conjunto de processos com estas restrições é bastante complexo, o que efetivamente impede o escalonamento destes em modo totalmente "on-line".

O objetivo deste trabalho é propor uma estratégia para o escalonamento de processos periódicos e esporádicos em STRC monoprocessados, sujeitos às restrições citadas acima. Para os processos periódicos, que possuem todos os seus parâmetros conhecidos de antemão, usa-se um algoritmo de escalonamento "off-line" ótimo, disponível na literatura. Para os processos esporádicos, que possuem tempos de pronto não determinísticos, uma vez que estes processos estão geralmente associados à eventos aleatórios do sistema cujos instantes de ocorrência são imprevisíveis, apresenta-se um procedimento inédito para o atendimento "on-line" destes.

O trabalho encontra-se organizado da seguinte forma: no capítulo 2 apresenta-se os conceitos fundamentais de escalonamento em STRC e faz-se uma revisão bibliográfica dos principais trabalhos disponíveis na literatura sobre escalonamento de processos em STRC monoprocessados; no capítulo 3 discute-se aspectos de escalonamento "off-line" e o algoritmo utilizado para o escalonamento de processos periódicos; o capítulo 4 descreve o procedimento proposto para o atendimento "on-line" de processos esporádicos, o qual utiliza informações obtidas do escalonamento gerado "off-line" para os processos periódicos; no capítulo 5 são discutidos aspectos do software implementado para a validação da estratégia de escalonamento proposta e apresentados exemplos de aplicações; finalmente, o capítulo 6 traz as conclusões, considerações finais e sugestões para o prosseguimento do trabalho.

CAPÍTULO 2

ESCALONAMENTO EM SISTEMAS DE TEMPO REAL CRÍTICO

ESCALONAMENTO EM SISTEMAS DE TEMPO REAL CRÍTICO

2.1 - INTRODUÇÃO

Em *Sistemas de Tempo Real* a correção do sistema depende de atividades logicamente corretas executadas oportunamente, ou seja, as atividades devem também satisfazer restrições temporais impostas pela aplicação. Os Sistemas de Tempo Real são divididos em duas grandes classes: *Sistemas de Tempo Real Não-Crítico* ("Soft Real-Time Systems) e *Sistemas de Tempo Real Crítico* - **STRC** ("Hard Real-Time Systems" - HRTS).

Nos Sistemas de Tempo Real Não-Crítico, as atividades devem ser executadas o mais rápido possível, mas não precisam terminar dentro de prazos específicos. Nestes sistemas, o objetivo é minimizar o tempo médio em que as atividades são executadas. Por outro lado, nos Sistemas de Tempo Real Crítico, as atividades devem ser executadas dentro de prazos específicos. Caso isto não ocorra, dependendo da aplicação, podem ocorrer consequências catastróficas.

Aplicações de Sistemas de Tempo Real Não-Crítico incluem: sistemas de reserva de passagens em companhias aéreas, sistemas de automação bancária, transações em bancos de dados, etc.. Para os Sistemas de Tempo Real Crítico, que são o objeto da atenção deste trabalho, destacam-se as aplicações em controle de processos industriais, usinas nucleares, sistemas de controle de vôo, telecomunicações, robótica e outras.

Um Sistema de Tempo Real consiste, tipicamente, em um *sistema controlador* e em um *sistema controlado*. Por exemplo, em uma fábrica automatizada, o sistema controlado é o chão de fábrica, com seus robôs, linhas de montagens, etc., enquanto o sistema controlador é o computador e as "interfaces" homem-máquina que gerenciam e coordenam as atividades no chão de fábrica. O sistema controlado pode ser visto como o ambiente com o qual o computador interage.

O sistema controlador interage com seu ambiente baseado na informação disponível sobre o sistema controlado. Esta informação, obtida a partir de sensores, deve ser consistente com o estado atual do ambiente pois, caso contrário, os efeitos das atividades do sistema controlador podem ser desastrosas. Portanto, faz-se necessário um monitoramento periódico do ambiente, bem como um processamento correto do ponto de vista lógico e temporal da informação sensoriada.

Devido às características do sistema controlado, o sistema controlador é constituído, geralmente, de diversas atividades distintas que necessitam ser executadas simultaneamente para poder acompanhar as mudanças de estado do sistema controlado. A realização de cada atividade do sistema controlador é feita através de um módulo de software específico chamado *processo.* Processos podem ser vistos como uma abstração de uma sequência de código onde, em qualquer ponto de suas computações, podem ser caracterizados pelas suas informações de estado, ou seja, contador de programa, pilha e variáveis estáticas[02].

2.2 - CARACTERÍSTICAS de PROCESSOS

Em **STRC** os diversos processos podem ser caracterizados sob vários aspectos, a saber: restrições de tempo, periodicidade, relações de precedência, relações de exclusão mútua, recursos, preempção e localização. A seguir, cada um destes aspectos é discutido brevemente.

2.2.1 - RESTRIÇÕES de TEMPO

As restrições de tempo de um processo podem ser especificadas através de vários parâmetros:

- *tempo de chegada* ("arrival time"), *a*: instante no qual o processo é invocado no sistema;
- *tempo de pronto* ("ready time, release time"), *r*:

instante mais cedo no qual o processo torna-se disponível para execução. O tempo de pronto de um processo é maior ou igual que seu tempo de chegada e, caso não existam restrições de recursos adicionais, temos $r = a$;

- *tempo de execução* ("computation time"), c: tempo máximo de execução de um processo. No cálculo deste parâmetro podem ser considerados alguns custos adicionais como, por exemplo, o custo de mudança de contexto;

- "*deadline*", d: prazo máximo para término de execução de um dado processo. d é um valor relativo ao tempo de chegada do processo. O "deadline" absoluto da é dado por: $da = a + d$. O "deadline" é a principal restrição de tempo de um processo.

2.2.2 - PERIODICIDADE

Dependendo da natureza do tempo de chegada dos processos estes podem ser classificados em:

- *processos periódicos*: processos que são invocados uma vez a cada intervalo de tempo fixo, ou seja, são invocados exatamente uma vez por período p. Todas as restrições de tempo de processos periódicos são conhecidas previamente e, em particular, os tempos de chegada podem ser pré-determinados. Exemplo típico destes são os processos que fazem varredura periódica de sensores. A figura 2.1 ilustra as diversas instâncias de um processo periódico P_i, onde o "deadline" coincide com o período, ou seja, $d_i = p_i$.

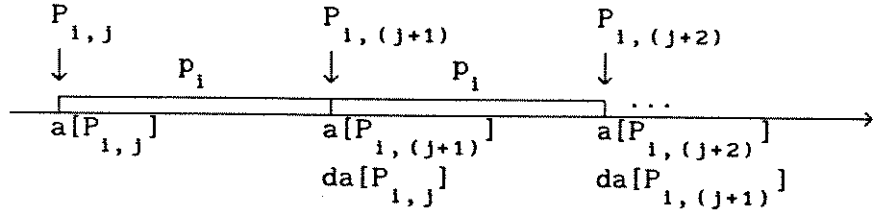


Figura 2.1 - Instâncias de um Processo Periódico

Nota-se que o "deadline" da $(j + 1)$-ésima instância do processo P_i, ou seja, processo $P_{i,(j+1)}$ coincide com a invocação da instância $(j + 2)$ e o "deadline" de qualquer instância dá-se p_i unidades de tempo após sua invocação, ou seja:

$$da[P_{i,j}] = a[P_{i,(j+1)}]$$
$$da[P_{i,j}] = a[P_{i,j}] + p_i.$$

As relações matemáticas acima são válidas apenas em situações onde o "deadline" d_i coincide com o período p_i. Em aplicações com restrições de tempo mais rígidas pode-se fazer necessário um "deadline" menor que o período. Obviamente, o "deadline" absoluto de qualquer processo P_i, independente da natureza de seu tempo de chegada, deve ser maior ou igual ao tempo de chegada do processo acrescido de seu tempo de execução:

$$da[P_i] \geq a[P_i] + c[P_i];$$

- *processos aperiódicos*: processos que podem ser invocados a qualquer instante, não possuindo tempos de chegadas determinísticos. Estes processos estão geralmente associados à eventos aleatórios, tais como aumento repentino de pressão, temperatura, etc., e cujos instantes de ocorrência são imprevisíveis. Processos aperiódicos que possuem prazos para término de execução ("deadline") são também chamados *processos esporádicos.*

2.2.3 - RELAÇÕES de PRECEDÊNCIA

Relações de precedência entre processos surgem quando um processo requer informações produzidas por outros processos. Neste caso, os processos deixam de ser independentes entre si, pois passa a existir uma ordem para execução destes. Quando um processo A precede um processo B, somente após o término de A o processo B poderá iniciar sua execução.

2.2.4 - RELAÇÕES de EXCLUSÃO MÚTUA

Quando um recurso qualquer do sistema (memória, arquivos, I/O, etc.) é compartilhado entre vários processos, faz-se necessário relações de exclusão mútua entre estes para impedir o acesso simultâneo ao recurso compartilhado, o que poderia ocasionar inconsistências. Uma das maneiras de assegurar-se exclusão mútua é o uso de primitivas do tipo semáforo[03].

2.2.5 - RECURSOS

Dependendo da atividade realizada por um processo, este pode necessitar de recursos específicos para poder executar, tais como unidades de I/O, processadores dedicados, etc. Neste caso, o tempo de pronto do processo é geralmente posterior ao tempo de chegada do mesmo, pois normalmente leva-se algum tempo para alocar todos os recursos necessários. Em muitos **STRC**, os recursos requeridos por um processo são assumidos estar disponíveis assim que o processo é invocado. Para estes sistemas, os processadores são os recursos primários.

2.2.6 - PREEMPÇÃO

Em **STRC** os processos podem ser *preemptíveis* ou *não-preemptíveis*, dependendo da natureza da aplicação. Um processo é dito preemptível se sua execução pode ser interrompida por outros processos de maior prioridade em qualquer instante, retornando sua execução mais tarde no ponto em que havia parado. Processos não-preemptíveis, uma vez iniciados, executam até o seu término ou até liberarem explicitamente o processador.

2.2.7 - LOCALIZAÇÃO

Quando no sistema existem múltiplos nós processadores, os processos devem ser alocados a estes considerando diversos aspectos, tais como o balanceamento de carga de processamento, minimização dos custos de comunicação, requisitos de tolerância à falhas e de recursos adicionais, etc..

O sistema controlador é constituído de diversos processos que necessitam ser executados simultâneamente para acompanhar o dinamismo do sistema controlado. Normalmente, o número de processos do sistema supera o número de processadores. Sempre que isto ocorre, surge a necessidade de um componente de software, o *escalonador*, que implementa um ou mais *algoritmos de escalonamento* para coordenar a execução dos processos.

2.3 - ALGORITMOS de ESCALONAMENTO

Um algoritmo de escalonamento é um conjunto de regras que determina o processo a ser executado em um particular momento. A seguir, apresenta-se uma definição mais formal e a classificação de algoritmos de escalonamento para **STRC**.

2.3.1 - DEFINIÇÃO e CLASSIFICAÇÃO de ALGORITMOS de ESCALONAMENTO para STRC

Processos são as entidades escalonáveis em **STRC**. A função de um algoritmo de escalonamento é determinar, para um dado conjunto de processos, se uma ordem (a sequência e os tempos) para execução dos processos existe, tal que as restrições de tempo, precedência, exclusão mútua, recursos e outras que por ventura existam sejam satisfeitas, e calcular tal ordem se ela existir[04]. Em muitas situações, o problema de encontrar um escalonamento para um conjunto de processos em **STRC** é bastante árduo devido à necessidade

de satisfazer os requisitos de tempo individuais dos processos (principalmente o "deadline"), os quais são ditados pelos fenômenos físicos controlados. Portanto, em **STRC**, o escalonamento de processos é o problema mais importante, porque é o algoritmo de escalonamento que garante que os processos cumprirão seus "deadlines". Uma taxonomia dos algoritmos de escalonamento para **STRC** é apresentada na figura 2.2[04].

Algoritmos de escalonamento para **STRC** podem ser *estáticos* ou *dinâmicos*. A abordagem estática calcula o escalonamento dos processos "off-line", e requer um conhecimento prévio completo das características dos processos. A abordagem dinâmica escalona os processos "on-line", e permite que processos possam ser invocados dinâmicamente. Abordagens estáticas, embora apresentem um baixo custo em tempo de execução, são inflexíveis e não se adaptam à ambientes cujo comportamento não são completamente previsíveis. As abordagens dinâmicas são mais flexíveis e podem se adaptar às mudanças do ambiente, mas apresentam um custo em tempo de execução maior.

De acordo com a arquitetura do sistema alvo, os algoritmos de escalonamento podem ser classificados em *centralizados* ou *distribuídos*. Um sistema centralizado é aquele no qual os processadores estão localizados em um único ponto do sistema, e o custo de comunicação inter-processadores é desprezível comparado com o custo de execução. Sistemas multiprocessadores com memória compartilhada e sistemas monoprocessadores são exemplos de sistemas centralizados. Já nos sistemas distribuídos, os processadores estão localizados em diferentes pontos do sistema, e o custo de comunicação inter-processadores é um importante fator que deve ser explicitamente levado em conta no escalonamento. Nesta última classe se enquadram, por exemplo, as redes locais de computadores.

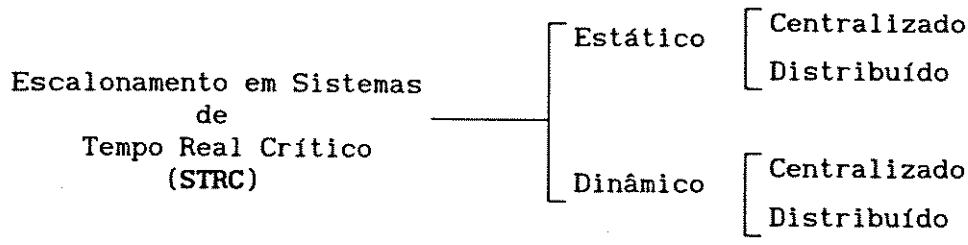


Figura 2.2 - Uma Taxonomia de Algoritmos de Escalonamento para STRC

2.3.2 - MEDIDAS de MÉRITO de ESCALONADORES

A escolha do processo a ser executado é feita pelo escalonador, o qual é um módulo do núcleo de tempo real ou do sistema operacional. Um escalonador deve coordenar o uso de todos os recursos do sistema usando um conjunto de algoritmos de escalonamento (obviamente no mínimo 1), que satisfaçam os seguintes objetivos[05], [06]:

- garantir que os processos cumpram seus "deadlines";
- fornecer bons tempos médios de resposta para processos aperiódicos que não possuem "deadline";
- garantir estabilidade em momentos de sobrecarga transitória do sistema. Quando o sistema está sobrecarregado por diversos eventos e satisfazer todos os "deadlines" é impossível, deve-se ainda garantir os "deadlines" de processos críticos previamente selecionados. Um escalonador com esta característica é dito ser *estável*;
- obter um alto *grau de escalonabilidade* (**G**). Dado um conjunto de n processos periódicos P_1, P_2, ..., P_n, onde cada processo P_i possui um tempo de execução c_i e um período p_i, definimos o *fator de utilização de processador* (**U**) pelos n processos como sendo:

$$U(n) = \sum_{i=1}^{n} (c_i/p_i).$$

O grau de escalonabilidade é um valor do fator de utilização de processador onde toda utilização de processador menor ou igual a este grau garante que os "deadlines" dos processos são cumpridos. O grau de escalonabilidade de um algoritmo é uma medida de sua utilidade, pois quanto maior for este grau, maior será o número de aplicações que poderão fazer uso dele.

A qualidade de escalonadores para **STRC** é julgada de acordo com os objetivos citados. Para um dado conjunto de processos, um algoritmo de escalonamento é dito ser *praticável*, se ele escalona os processos de modo que todos cumpram seus "deadlines". Uma condição necessária para que exista um escalonamento praticável para um conjunto arbitrário de processos é que o fator de utilização **U** não seja maior que o número de processadores disponíveis.

Diz-se que um algoritmo de escalonamento dinâmico *garante* um processo recentemente chegado se o algoritmo pode encontrar uma nova ordem para os processos previamente garantidos e o novo processo, tal que todos os processos cumpram seus "deadlines". A principal medida de desempenho para um algoritmo de escalonamento dinâmico é a *taxa de garantia*, que é o número total de processos garantidos dividido pelo número total de processos que chegaram no sistema.

Um algoritmo de escalonamento estático é dito ser *ótimo* se, para qualquer conjunto de processos, ele sempre produz uma ordem para execução que satisfaz as restrições dos processos, todas as vezes que algum outro algoritmo possa fazê-lo também. Um algoritmo de escalonamento dinâmico é dito ser *ótimo* se ele sempre produz uma ordem para execução dos processos praticável, todas as vezes que um algoritmo de escalonamento estático com completo conhecimento prévio de todas as restrições dos processos também possa fazê-lo. Assim, um algoritmo de escalonamento dinâmico ótimo maximiza a taxa de garantia dos processos que chegam ao sistema[04].

A seguir, discute-se os principais trabalhos relativos ao escalonamento de processos para **STRC** monoprocessados.

2.3.3 - ESCALONAMENTO de PROCESSOS para STRC MONOPROCESSADOS

Em **STRC**, como já foi dito, a correção do sistema não depende apenas do resultado lógico da computação, mas também dos instantes nos quais estes resultados são produzidos. Procura-se obter a correção lógica do sistema através de técnicas adequadas de especificação, projeto e verificação, enquanto a teoria de escalonamento busca atender o cumprimento dos requisitos temporais especificados.

Liu e Layland[01] apresentaram dois algoritmos clássicos de escalonamento para **STRC** monoprocessados: Taxa Monotônica e o "Earliest Deadline".

2.3.3.1 - ALGORITMO TAXA MONOTÔNICA

O algoritmo estático Taxa Monotônica foi introduzido para o escalonamento de processos periódicos preemptivos, independentes (sem restrições de precedência e exclusão mútua), com "deadline" igual ao período, e que não necessitam de recursos específicos para executar[01]. Prioridades fixas são atribuídas aos processos baseado em suas taxas de requisição, ou seja, quanto mais requisitado um processo (i.e., menor o seu período) maior será a sua prioridade.

Na figura 2.3 é apresentado um exemplo de escalonamento dos processos A e B usando-se o algoritmo Taxa Monotônica. Inicialmente, em modo "off-line", atribui-se aos processos A e B, respectivamente, as prioridades 1 e 2, pelo fato do período do processo A ser menor que o do processo B. A sequência de execução dos processos pode ser acompanhada através do diagrama de Gantt.

Processos Periódicos	c	p	d	Prioridade
A	4	8	8	1
B	5	16	16	2

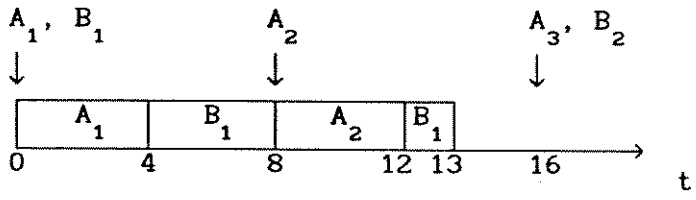


Figura 2.3 - Exemplo de Taxa Monotônica

O processo em execução em qualquer instante é o processo pronto com maior prioridade. Em $t = 0$, os processos A e B iniciam seu primeiro período. O processo A, por possuir a maior prioridade, é escalonado e ocupa o processador até $t = 4$, quando completa a execução de sua primeira instância (A_1). Em $t = 4$ o processo B_1 é escalonado, por ser o processo pronto de maior prioridade, e executa 4 unidades até $t = 8$, quando chega a segunda instância do processo A. O processo B, por possuir uma prioridade menor que a de A, sofre preempção e dá lugar ao processo A que executa até $t = 12$, quando é completada sua segunda instância. Após isso, o processo B retoma a posse do processador e executa de $t = 12$ a $t = 13$, completando a sua primeira instância.

Em [01], Liu e Layland provaram que o algoritmo Taxa Monotônica é ótimo, no sentido que nenhuma outra regra de associação de prioridades fixas pode escalonar um conjunto de processos que não pode ser escalonado pelo algoritmo Taxa Monotônica. Eles também mostraram que o grau de escalonabilidade do algoritmo Taxa Monotônica para um conjunto de n processos periódicos, com as características citadas no início desta sub-seção é:

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

Portanto, dado um conjunto de n processos periódicos, sempre que

$\mathbf{U}(n) \leq \mathbf{G}(n)$, tem-se a garantia de que os processos cumprirão seus "deadlines".

Para o exemplo anterior (figura 2.3), temos o seguinte valor do fator de utilização de processador:

$$\mathbf{U}(2) = \sum_{i=1}^{2} (c_i/p_i) = 4/8 + 5/16 = 0,8125;$$

o grau de escalonabilidade para 2 processos é:

$$\mathbf{G}(2) = 2(2^{1/2} - 1) = 0,828.$$

Como (0,8125 < 0,828), temos a garantia de que todas as instâncias dos processos periódicos A e B sempre cumprirão seus "deadlines".

O grau de escalonabilidade **G**(n), quando o número de processos n tende para o infinito, converge para $ln\ 2$ (= 69%). A condição $\mathbf{U}(n) \leq n(2^{1/n} - 1)$, que garante que os n processos sempre cumprirão seus "deadlines", é uma condição suficiente, pois o grau de escalonabilidade $\mathbf{G}(n)$ é derivado a partir de uma situação de pior caso, bastante pessimista e difícil de acontecer na prática. Lehoczky, Sha e Ding[07] desenvolveram um teste de escalonabilidade exato para o algoritmo Taxa Monotônica (condição necessária e suficiente) que pode ser aplicado quando o teste $\mathbf{U}(n) \leq \mathbf{G}(n)$ falha.

O algoritmo Taxa Monotônica é de grande importância, pois ele pode ser usado como base para o desenvolvimento de uma família de algoritmos que atendem uma grande variedade de problemas práticos. O algoritmo Taxa Monotônica apresenta as seguintes qualidades para o escalonamento em **STRC**:

- *Alto fator de utilização de processador*: embora o grau de escalonabilidade de 69% seja baixo, ele é pessimista e representa o pior caso possível. Em [07], através de análise estocástica para um conjunto de processos periódicos gerados aleatoriamente, escalonados pelo algoritmo Taxa Monotônica, chegou-se à conclusão que a média do fator de utilização é muito melhor que o pior caso. Concluiu-se que uma boa aproximação seria 88%, com alguns sistemas atingindo até 99%.

• *Estabilidade sob sobrecarga transitória*: através de um método de transformação de período para o algoritmo Taxa Monotônica[08], este pode garantir que os "deadlines" de processos muito críticos, mas que possuem longos períodos, o que acarretaria em baixas prioridades, possam ser satisfeitos em situações de sobrecarga transitória do sistema.

• *Processos aperiódicos*: **STRC** normalmente possuem processos periódicos e aperiódicos. Vários algoritmos compatíveis com o Taxa Monotônica, entre eles o "Deferrable Server"[05], [09] e o "Sporadic Server"[05], foram desenvolvidos para fornecer bons tempos médios de resposta para processos aperiódicos e/ou garantir os "deadlines" de processos esporádicos e ainda satisfazer os "deadlines" de processos periódicos, previamente garantidos. Estes algoritmos usam um processo especial, chamado *servidor aperiódico*, para atender as requisições de processos aperiódicos.

• *Compartilhamento de recursos*: quando processos compartilham recursos, faz-se necessário o uso de primitivas de sincronização do tipo semáforo, por exemplo, para assegurar a consistência no uso do recurso compartilhado. A aplicação direta deste mecanismo de sincronização pode provocar um problema, chamado de *inversão de prioridades*, que ocorre quando um processo de alta prioridade é forçado a esperar a execução de vários processos de menor prioridade por um período de tempo indefinido, afetando a previsibilidade do sistema. Os *protocolos de herança de prioridades*[10], [11] limitam o problema de inversão de prioridades. Usando-se este protocolo, no máximo um processo de baixa prioridade pode bloquear um processo de maior prioridade e, ainda, tem-se assegurado a ausência de "deadlocks", que poderiam ocorrer quando processos realizam acessos aninhados às regiões críticas. Também é apresentado um conjunto de condições suficientes sob as quais um conjunto de processos periódicos que compartilham recursos usando este protocolo podem ser escalonados pelo algoritmo Taxa Monotônica. Em [12] é apresentada uma extensão do protocolo para sistemas multiprocessadores com memória compartilhada.

• *Baixo custo de escalonamento*: uma vez que o algoritmo atribui prioridades estáticas aos processos, a seleção do

processo que irá executar em um dado momento é muito simples. Esta qualidade é comum a todos os algoritmos de escalonamento estáticos.

Como podemos ver, talvez a principal qualidade do algoritmo Taxa Monotônica é a sua *extensibilidade*, pois ele nos permite um grande número de extensões além do tradicional escalonamento de processos periódicos independentes. Além do que já foi dito, o algoritmo Taxa Monotônica pode ser convenientemente usado para escalonar processos onde *computação imprecisa* é permitida[13], [14]. Permite ainda a inclusão e exclusão de processos durante a execução do sistema e a alteração de parâmetros de processos, usando-se o "Mode Change Protocol"[15]. Em [16] e [06] são apresentadas compilações da teoria de escalonamento baseada no algoritmo Taxa Monotônica e exemplos de aplicações.

2.3.3.2 - ALGORITMO "EARLIEST DEADLINE"

O algoritmo dinâmico "Earliest Deadline" foi introduzido para o escalonamento de processos periódicos preemptivos, independentes (sem restrições de precedência e exclusão mútua), com "deadline" igual ao período, e que não necessitam de recursos específicos para executar[01]. Prioridades são atribuídas dinamicamente aos processos (prioridades não fixas) de acordo com os "deadlines" dos processos. Em qualquer instante de tempo t, o processo de maior prioridade é o processo pronto com "deadline" absoluto mais próximo de t.

Na figura 2.4 é apresentado um exemplo de escalonamento dos processos A e B usando-se o algoritmo "Earliest Deadline". A sequência de execução dos processos pode ser acompanhada através do diagrama de Gantt.

Processos Periódicos	c	p	d
A	3	6	6
B	9	18	18

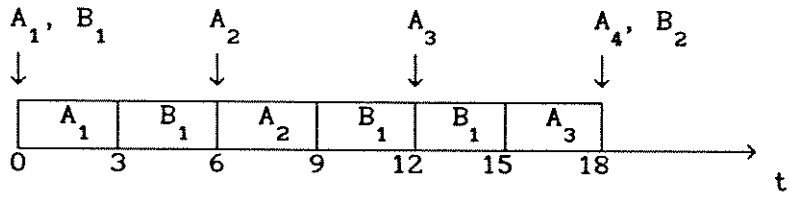


Figura 2.4 - Exemplo de "Earliest Deadline"

O processo em execução em qualquer instante é o processo pronto com maior prioridade. Em t = 0 os processos A e B iniciam seu primeiro período. O processo A, por possuir o "deadline" mais próximo ($da[A_1] = 6$), é escalonado e executa até t = 3, quando é completada a execução de sua primeira instância. Neste instante, sendo B o único processo pronto, ele entra em execução e executa 3 unidades até t = 6, quando chega uma nova instância do processo A. Como o "deadline" da segunda instância do processo A ($da[A_2] = 12$) é menor que o "deadline" da instância corrente do processo B ($da[B_1] = 18$), o processo B sofre preempção e dá lugar ao processo A que executa até t = 9, quando é completada sua segunda instância. Após isso, o processo B retoma a posse do processador e executa até t = 12, quando chega a terceira instância do processo A. Como o "deadline" desta nova instância de A ($da[A_3] = 18$) é igual ao "deadline" da instância corrente de B, o processo B não sofre preempção, permanecendo em execução até t = 15, quando é completado. A terceira instância do processo A é executada de t = 15 até t = 18.

Liu e Layland[01] provaram que o algoritmo "Earliest Deadline" é ótimo. Eles mostraram também que o grau de escalonabilidade do algoritmo "Earliest Deadline" para um conjunto de n processos periódicos, com as características citadas no início desta sub-seção é:

$$G(n) = 1.$$

Portanto, dado um conjunto de n processos periódicos, sempre que $U(n) \leq 1$, tem-se a garantia de que os processos cumprirão seus "deadlines". A condição $U(n) \leq 1$ é uma condição necessária e suficiente. Assim, utilizando-se o algoritmo "Earliest Deadline", pode-se obter uma completa utilização do processador ($U = 1$), fato verificado no exemplo da figura 2.4.

Mok[02], [17] propôs o monitor "kernelized", baseado no algoritmo "Earliest Deadline", que permite escalonar processos que realizam trocas de mensagens síncronas ("rendezvous") e que também possuem relações de exclusão mútua. O monitor "kernelized" garante a exclusão mútua alocando o processador somente em intervalos de tempo ininterruptíveis, denominados quantum, definidos de acordo com o tamanho da maior região crítica dos processos. Assim, como o quantum é a quantidade mínima de tempo de processamento que um processo pode receber, o monitor "kernelized" só se aplica efetivamente a processos que possuam regiões críticas bem pequenas. Em [18] é apresentado um protocolo de herança de prioridades dinâmico, o qual permite a utilização de semáforos para assegurar exclusão mútua entre processos escalonados pelo algoritmo "Earliest Deadline", sem incorrer no problema de inversão de prioridades. Também é apresentado uma condição suficiente, que permite verificar se o algoritmo "Earliest Deadline" é praticável para um conjunto de processos que compartilham recursos usando este protocolo.

O escalonamento de processos para **STRC**, especialmente em ambientes monoprocessados, tem recebido bastante atenção por parte da comunidade científica. Hsieh[19] estudou a problemática de escalonamento de processos em diversos ambientes, baseado nos

algoritmos Taxa Monotônica e "Earliest Deadline", e fez propostas para a implementação destes algoritmos sobre núcleos de tempo real convencionais. Spanó[20] desenvolveu uma ferramenta de software para a análise de algoritmos de escalonamento para STRC monoprocessados. A ferramenta traz os algoritmos clássicos da literatura embutidos e também permite ao usuário definir novas políticas de escalonamento, através do uso de uma biblioteca de rotinas. A ferramenta possui diversas facilidades para o acompanhamento da simulação de escalonamento e análise dos resultados.

Em STRC com alguma complexidade os processos a serem escalonados estão sujeitos a um grande número de restrições: tempo de pronto, "deadline", relações de precedência e relações de exclusão mútua. O problema de escalonar um conjunto de processos com estas restrições é conhecido ser "NP-hard"[21], [22], o que efetivamente impede o escalonamento destes em modo "on-line", fazendo-se necessário técnicas de escalonamento "off-line".

2.3.3.3 - ALGORITMOS de ESCALONAMENTO "OFF-LINE"

Em aplicações reais de STRC frequentemente depara-se com complexos problemas de escalonamento, nos quais estão envolvidos um número relativamente grande de processos com rígidas restrições temporais e alto fator de utilização de CPU, além de intrincadas relações de interdependência entre estes. Neste caso, o emprego de práticas correntes de desenvolvimento de sistemas de tempo real, tais como: o uso de escalonadores preemptivos orientados a prioridades estáticas; o uso de complexos e não determinísticos mecanismos de sincronização em tempo de execução; a permissão para que eventos interrompam o processador e ocupem os recursos do sistema em instantes aleatórios, além do uso de técnicas de simulação estocástica para validar os requisitos do sistema, devem ser evitadas, uma vez que estas comprometem a previsibilidade do sistema, além de incorrerem em altos custos em tempo de execução e de frequentemente falharem em encontrar um escalonamento praticável para um conjunto de processos, mesmo quando um exista[23].

A maioria dos problemas de escalonamento em **STRC** possui complexidade bastante alta, tipicamente "NP-hard"[22], [23]. Esta característica, aliada ao fato de que a quantidade de tempo disponível para um escalonador "on-line" computar escalonamentos dinamicamente ser bastante escassa, frequentemente faz com que a única maneira prática de satisfazer as rígidas restrições temporais em **STRC**, de modo a assegurar a previsibilidade do sistema, seja o uso de técnicas de escalonamento "off-line". Xu e Parnas[23] apresentaram um guia dos algoritmos para problemas de escalonamento matemático e analisaram quais destes algoritmos podem ser usados para o escalonamento em **STRC**.

2.4 - CONSIDERAÇÕES FINAIS

O algoritmo Taxa Monotônica, na categoria de escalonadores estáticos e o "Earliest Deadline", na categoria de escalonadores dinâmicos, são comprovadamente ótimos[01], no sentido que sempre que existir algum escalonamento praticável para um conjunto de processos, estes algoritmos encontram uma solução. Para ambos existem expressões matemáticas que permitem verificar a escalonabilidade de um conjunto de processos mediante um simples cálculo[01], [07]. Contudo, estes algoritmos somente são ótimos em um cenário bastante restrito: ambiente monoprocessador, processos periódicos preemptíveis com "deadline" igual ao período e independentes (sem restrições de precedência e exclusão mútua).

No próximo capítulo serão discutidos aspectos de escalonamento "off-line" e apresentado um algoritmo "off-line" ótimo, disponível na literatura, capaz de escalonar processos sujeitos a restrições de tempo de pronto, "deadline" e arbitrárias relações de precedência e exclusão mútua.

CAPÍTULO 3

UM ESCALONADOR "OFF-LINE"

PARA

SISTEMAS DE TEMPO REAL CRÍTICO

UM ESCALONADOR "OFF-LINE" PARA SISTEMAS DE TEMPO REAL CRÍTICO

3.1 - INTRODUÇÃO

Até recentemente não existia na literatura algoritmo para encontrar um escalonamento ótimo no caso de um conjunto de processos sujeitos a restrições de tempo de pronto, "deadline" e relações arbitrárias de precedência e exclusão mútua[24]. Os projetistas de **STRC** eram obrigados a fazer o escalonamento "off-line" através de métodos "ad hoc", os quais são bastante susceptíveis a erros e frequentemente falham em encontrar, mesmo quando existe, um escalonamento.

Xu e Parnas[21] apresentaram um algoritmo para encontrar, todas as vezes que existir, um escalonamento no caso de um conjunto de processos preemptivos em um ambiente monoprocessador, tal que cada processo inicie sua execução após seu tempo de pronto e termine no máximo até o seu "deadline", satisfazendo todas as relações de precedência e exclusão mútua, definidas sobre pares ordenados de segmentos de processos. O algoritmo tem como objetivo encontrar um escalonamento sempre que uma solução existir. Este requisito, juntamente com o fato do problema ser "NP-hard", efetivamente permite apenas soluções do tipo enumerativa, a menos que **P = NP**[22]. O algoritmo faz uso de uma técnica de enumeração implícita "branch and bound", que na maioria dos casos apresenta um desempenho muito melhor que uma enumeração completa, mas possui complexidade exponencial[25]. Com o uso do método "branch and bound", apesar de se considerar todas as possibilidades de escalonamento, isto não é feito de modo explícito, pois o método explora a árvore de busca inteligentemente, fazendo uso da idéia de "bounding"[25].

A utilização de um algoritmo de escalonamento "off-line" para **STRC** pode reduzir significativamente os custos de

escalonamento e mudanças de contexto em tempo de execução. Mais importante que isto, todas as relações de precedência e exclusão mútua já estão asseguradas no escalonamento obtido "off-line", não sendo necessário o uso de primitivas de sincronização em tempo de execução, e tem-se ainda a garantia prévia de que os "deadlines" serão satisfeitos. Isto aumenta a previsibilidade do sistema e diminui a possibilidade de "deadlocks". A seguir introduz-se definições e notações usadas pelo algoritmo "off-line" de Xu e Parnas[21].

3.2 - DEFINIÇÕES e NOTAÇÕES

O *conjunto de processos* a ser escalonado será denotado por $\mathfrak{P}$.

Cada processo $p \in \mathfrak{P}$ consiste de uma sequência finita de *segmentos* $p[0]$, $p[1]$, ..., $p[n[p]]$, onde $p[0]$ é o primeiro segmento e $p[n[p]]$ é o último segmento do processo p.

Para cada segmento i define-se:

- um tempo de pronto $r[i]$;
- um "deadline" absoluto $da[i]$;
- um tempo de execução $c[i]$.

Assume-se que os parâmetros $r[i]$, $da[i]$, $c[i]$ possuem valores inteiros não negativos, dados em múltiplos de uma unidade de tempo básica do sistema.

O *conjunto de todos os segmentos* pertencentes a processos em $\mathfrak{P}$ será denotado por $\mathbf{S}(\mathfrak{P})$.

O algoritmo de Xu e Parnas[21] é bastante flexível, pois permite modelar situações reais onde algumas porções de um processo são preemptíveis por certas porções de outros processos, enquanto outras porções do processo não são preemptíveis por certas porções de outros processos. Essas porções de processos correspondem aos segmentos. Inicialmente, todos os processos de $\mathfrak{P}$ devem ser decompostos em segmentos para se obter o conjunto $\mathbf{S}(\mathfrak{P})$, que constitui a entrada do algoritmo, juntamente com as relações de precedência e

exclusão mútua definidas entre pares ordenados de segmentos de $\mathbf{S}(\mathfrak{P})$.

A partir do tempo de pronto, tempo de execução e "deadline" absoluto de cada processo e do tempo de execução e tempo de início de cada segmento relativo ao início do processo contendo o segmento, pode-se calcular o tempo de pronto, tempo de execução e "deadline" absoluto para cada segmento. Como exemplo, consideremos um processo $A \in \mathfrak{P}$ com tempo de pronto r igual a 2, tempo de execução c igual a 7 e "deadline" absoluto da igual a 12. O processo A acessa um recurso 1 qualquer, compartilhado com outros processos do sistema, em sua porção intermediária de tamanho 3. A decomposição do processo em segmentos é apresentada na figura 3.1. Inicialmente, faz-se o "deadline" do último segmento como sendo o "deadline" do processo e o tempo de pronto do primeiro segmento igual ao tempo de pronto do processo. Os demais valores de parâmetros são obtidos de modo direto.

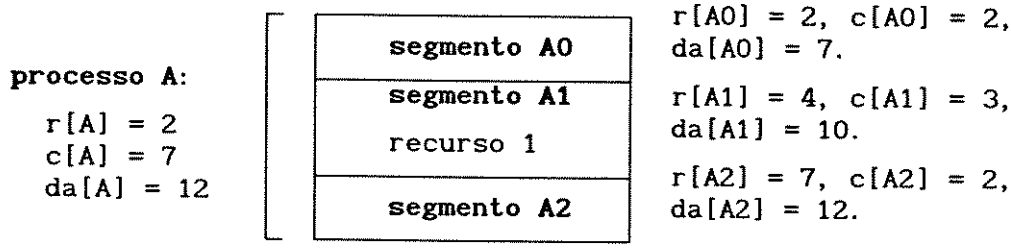


Figura 3.1 - Decomposição de um Processo em Segmentos

Cada segmento i consiste de uma sequência de *unidades de segmento* $(i, 0), (i, 1), \ldots, (i, c[i] - 1)$, onde $(i, 0)$ é a primeira unidade de segmento e $(i, c[i] - 1)$ é a última unidade de segmento do segmento i. Intuitivamente, uma unidade de segmento é a menor porção indivisível de um processo. Cada unidade de segmento requer uma unidade de tempo para executar, durante a qual ele não pode ser preemptado por qualquer outro processo. O *conjunto de unidades de segmento de* $\mathbf{S}(\mathfrak{P})$ será denotado por $\mathbf{U}$ e definido como:

$$\mathbf{U} = \{(i, k) \mid i \in \mathbf{S}(\mathfrak{P}) \wedge 0 \leq k \leq (c[i] - 1)\}.$$

Um *escalonamento* de um conjunto de processos $\mathfrak{P}$ é uma função Π: $\mathbf{U} \longrightarrow [0, \infty)$, satisfazendo as seguintes propriedades:

1) $\forall\ t \in [0, \infty)$: $|\{(i, k) \in \mathbf{U} \mid \Pi(i, k) = t\}| \leq 1$;

2) $\forall\ (i, k_1), (i, k_2) \in \mathbf{U}$:
$(k_1 < k_2) \Rightarrow (\Pi(i, k_1) < \Pi(i, k_2))$;

3) $\forall\ p \in \mathfrak{P}, \forall\ i, j, 0 \leq i, j \leq n[p]$:
$(i < j) \Rightarrow (\Pi(p[i], c[p[i]] - 1) < \Pi(p[j], 0))$.

A condição 1) estabelece que não mais que um segmento pode estar executando em qualquer instante. A condição 2) estabelece que um escalonamento deve preservar a ordem das unidades de segmento em cada segmento. A condição 3) estabelece que um escalonamento de $\mathfrak{P}$ deve preservar a ordem dos segmentos em cada processo.

Diz-se que o *segmento i executa em t* se e somente se:

$\exists\ k, 0 \leq k \leq (c[i] - 1)$: $\Pi(i, k) = t$.

Define-se o *tempo de início de execução* do segmento i como sendo:

$s[i] = \Pi(i, 0)$.

Define-se o *tempo de término de execução* do segmento i como sendo:

$e[i] = \Pi(i, c[i] - 1) + 1$.

O *retardo("lateness")* *de um segmento* i em um escalonamento de $\mathfrak{P}$ é definido como:

$l[i] = e[i] - da[i]$.

O *retardo de um escalonamento de* $\mathfrak{P}$ é definido como sendo o valor dado por:

$max\{l[i] \mid i \in \mathbf{S}(\mathfrak{P})\}$.

Define-se *segmento mais retardado* como sendo o segmento que realiza o valor do retardo de um escalonamento de $\mathfrak{P}$.

Um *escalonamento válido* de um conjunto de processos $\mathfrak{P}$ é um escalonamento de $\mathfrak{P}$ satisfazendo as seguintes propriedades:

$\forall\ i, j \in \mathbf{S}(\mathfrak{P})$:

1) $s[i] \geq r[i]$;

2) $(i\ \mathbf{PC}\ j) \Rightarrow (e[i] \leq s[j])$;

3) $(i\ \mathbf{EX}\ j \wedge s[i] < s[j]) \Rightarrow (e[i] \leq s[j])$.

Acima, a condição 1) estabelece que cada processo pode somente iniciar execução após seu tempo de pronto. A condição 2) estabelece que em um escalonamento válido, se um segmento i precede um segmento j(i **PC** j), então, em qualquer circunstância, o segmento j não pode iniciar sua execução antes do segmento i ter completado sua execução. A condição 3) estabelece que em um escalonamento válido, se um segmento i exclui um segmento j(i **EX** j), então o segmento j não pode preemptar o segmento i, ou seja, se o segmento i iniciou execução antes do segmento j, então o segmento j pode somente iniciar execução após o segmento i ter completado sua execução.

Os conjuntos de relações de precedência e exclusão são iniciados com as relações de precedência e exclusão mútua que devem ser satisfeitas no problema original. Em adição, a fim de garantir a ordenação dos segmentos dentro de cada processo, faz-se:

$$(p[k] \textbf{ PC } p[k+1]),\ \forall\ p \in \mathfrak{P},\ 0 \le k \le n[p] - 2.$$

Logo, um escalonamento válido satisfaz todas as restrições de tempo de pronto, exclusão e precedência do problema original.

Um *escalonamento praticável* de um conjunto de processos $\mathfrak{P}$ é um escalonamento válido de $\mathfrak{P}$ tal que o valor de seu retardo é menor ou igual a zero, ou seja, todos os segmentos de $\mathbf{S}(\mathfrak{P})$ cumprem seus "deadlines".

Um *escalonamento ótimo* de um conjunto de processos $\mathfrak{P}$ é um escalonamento válido de $\mathfrak{P}$ com mínimo retardo. Portanto, o critério de otimização do algoritmo é a minimização do retardo do escalonamento.

Uma maneira de reduzir o retardo de um segmento, na tentativa de fazer com que este cumpra seu "deadline", é forçar o segmento a preemptar um segmento escalonado antes dele. Para forçar a preempção entre dois segmentos, uma restrição de sincronização entre pares ordenados de segmentos, denotada por **PM**, é definida:

$\forall\ i,\ j \in \mathbf{S}(\mathfrak{P})$:

- $(i\ \mathbf{PM}\ j) \wedge (r[i] \leq r[j]) \Rightarrow s[j] \geq e[i]$;
- $(i\ \mathbf{PM}\ j) \wedge (r[j] < r[i]) \Rightarrow$ segmento j *não executa* no intervalo $[r[i],\ e[i])$.

Inicialmente, o conjunto de relações **PM** é vazio. Novas relações **PM** vão sendo adicionadas ao conjunto pelo algoritmo na tentativa de minimizar o retardo do escalonamento de $\mathfrak{P}$.

Após a decomposição dos processos em segmentos, cada segmento do conjunto $\mathbf{S}(\mathfrak{P})$ possui um tempo de pronto, os quais são ditados pelas características do sistema controlado. A fim de melhorar a eficiência do algoritmo, o tempo de pronto de cada segmento precedido ou preemptado por outro segmento, é revisado antes do cálculo do escalonamento em cada nó da árvore de busca, obtendo-se um *tempo de pronto ajustado.* Se uma relação de precedência do tipo $(i\ \mathbf{PC}\ j\ )$ existe, então o tempo de pronto do segmento j é ajustado para o tempo de término mais cedo possível do segmento i, ou seu próprio tempo de pronto, caso este seja maior. Do mesmo modo, se uma relação de preempção $(i\ \mathbf{PM}\ j)$ existe e a relação $(r[i] \leq r[j])$ segue, então o tempo de pronto do segmento j será ajustado do mesmo modo citado acima[21], [26]. O tempo de pronto ajustado de um segmento j é denotado como $r'[j]$ e determinado recursivamente da seguinte forma:

$\forall\ i,\ j \in \mathbf{S}(\mathfrak{P})$:

$$r'[j] = max(\{r'[i] + c[i] \mid (i\ \mathbf{PC}\ j)\} \cup \{r'[i] + c[i] \mid (i\ \mathbf{PM}\ j) \wedge (r[i] \leq r[j])\} \cup \{r[j]\}).$$

O "deadline" de cada segmento pode ser também revisado, a fim de melhorar o desempenho do algoritmo, obtendo-se um *"deadline"*

ajustado[26]. Um segmento que precede outros segmentos tem seu "deadline" ajustado para o tempo de término mais tardio possível que permita que os segmentos precedidos por este cumpram seus "deadlines", ou para seu próprio "deadline", caso este seja menor. O "deadline" ajustado de um segmento j é denotado como $da'[j]$ e determinado da seguinte forma:

$$\forall\ j,\ k \in \mathbf{S}(\mathfrak{P}):$$
$$da'[j] = min(\{da'[k] - c[k] \mid (j\ \mathbf{PC}\ k)\} \cup \{da[j]\}).$$

A revisão dos "deadlines" pode ser executada apenas uma única vez, durante os procedimentos de iniciação do algoritmo. Em [19] Hsieh analisa com detalhes o problema de revisão de "deadlines", apresentando um programa para automatizar a tarefa.

Ao calcular um escalonamento, o algoritmo determina em todo instante t um conjunto de segmentos que são elegíveis para executar. Em qualquer instante t, $t \in [0, \infty)$, diz-se que o *segmento j é elegível em t* se e somente se:

1) $(t \geq r'[j]) \wedge \neg(e[j] \leq t)$;
2) $\neg(\exists\ i:\ (i\ \mathbf{PC}\ j) \wedge \neg(e[i] \leq t))$;
3) $\neg(\exists\ i:\ (i\ \mathbf{EX}\ j) \wedge (s[i] < t) \wedge \neg(e[i] \leq t))$.

A definição acima garante que, se o segmento j é elegível em t, então j pode ser colocado em execução em t sem violar as propriedades de um escalonamento válido.

Uma *solução válida* para um conjunto de processos $\mathfrak{P}$ é um escalonamento válido de $\mathfrak{P}$ satisfazendo as seguintes propriedades:

$\forall\ t \in [0, \infty)$:

1) $\forall\ j$: ($\exists\ i$: (i **PM** $j \wedge i$ *é elegível em* t) $\vee$
($da[i] < da[j] \wedge \neg(j$ **PM** $i) \wedge$
i *é elegível em* t) $\vee$
($da[i] = da[j] \wedge c[i] > c[j] \wedge$
$\neg(j$ **PM** $i) \wedge i$ *é elegível em* t))
$\Rightarrow j$ *não executa em* t;

2) $\exists\ i$: i *é elegível em* t
$\Rightarrow \exists\ i$: i *executa em* t.

Em adição a um escalonamento válido, que satisfaz restrições de tempo de pronto, exclusão e precedência, uma solução válida também satisfaz restrições de *prioridades de execução*, definidas pelo conjunto de relações **PM** e "deadlines" dos segmentos. Acima, a condição 1) estabelece que em uma solução válida, se pelo menos um segmento i preempta o segmento j(i **PM** j) e i *é elegível em* t; ou se pelo menos um segmento i possui um "deadline" menor que j e i *é elegível em* t e j não preempta o segmento $i(\neg(j$ **PM** $i))$; ou se pelo menos um segmento i possui um "deadline" igual mas um tempo de execução maior que j e i *é elegível em* t e j não preempta o segmento $i(\neg(j$ **PM** $i))$, então o segmento j não pode executar no instante t. A condição 2) estabelece que em uma solução válida, em qualquer instante t, se pelo menos um segmento está *elegível*, então um segmento deve executar em t. A condição 2) garante que todos os segmentos de $\mathbf{S}(\mathfrak{P})$ eventualmente completarão suas execuções em uma solução válida desde que todas as relações definidas sobre pares ordenados de segmentos sejam consistentes.

Algumas combinações de restrições de precedência, exclusão e preempção, definidas sobre pares ordenados de segmentos, são *inconsistentes*. Na figura 3.2, cada par de restrições indicadas por um **X** são definidas inconsistentes. Todos os demais pares de relações são consistentes.

	i PC *j*	*j* PC *i*	*i* EX *j*	*j* EX *i*	*i* PM *j*	*j* PM *i*
i PC *j*		X				X
i EX *j*						X
i PM *j*		X		X		X

Figura 3.2 - Restrições de Sincronização Inconsistentes

Um procedimento simplificado para gerar uma solução válida em cada nó da árvore de busca é apresentado na figura 3.3. Iniciando em $t = 0$, e em cada subsequente incremento de t, o procedimento determina qual segmento do conjunto de segmentos $\{j \mid j$ *é elegível em* $t \wedge \neg(\exists\ i$: i *é elegível em* $t \wedge i$ **PM** $j)\}$ será escalonado, usando uma estratégia "earliest deadline". No apêndice A tem-se uma versão do algoritmo em pseudo-código.

Para encontrar-se uma solução válida praticável ou ótima usa-se uma técnica de enumeração implícita "branch and bound". O algoritmo define uma árvore de busca que tem em seu nó raiz uma solução válida inicial que satisfaz todas as relações de precedência e exclusão especificadas no problema original. Cada nó da árvore de busca, tal como a raiz, corresponde a uma solução válida completa, mas não necessariamente praticável. A seguir, veremos como melhorar uma solução válida, na tentativa de se obter um escalonamento praticável ou ótimo.

```
t := 0
Enquanto ¬(∀ i: e[i] ≤ t) faça
início
  Se (∃ i: t = r'[i] ∨ t = e[i])
  então
  início
    • No conjunto {j | j é elegível em t ∧
      ¬(∃ i: i é elegível em t ∧ i PM j)}
      selecione o segmento j que tenha mínimo
      da[j]. Em caso de empate, selecione o
      segmento j que possua máximo c[j].
    • Coloque j em execução.
  fim
  t := t + 1
fim
```

Figura 3.3 - Procedimento Simplificado para Calcular uma Solução Válida

3.3 - MELHORANDO uma SOLUÇÃO VÁLIDA

Quando uma solução válida é gerada em um nó da árvore de busca, caso esta não seja praticável, ela contém um certo número de segmentos retardados, ou seja, segmentos cujo valor de retardo é maior do que zero. Deve-se então expandir o nó em nós filhos. Cada nó filho representa uma diferente maneira de reduzir o retardo do segmento mais retardado da solução válida de seu nó pai.

Da definição de retardo de um segmento j ($l[j] = e[j] - da[j]$), pode-se observar que uma redução do retardo é somente possível se o segmento j terminar sua execução mais cedo, isto é, reduzir $e[j]$. Portanto, para reduzir o retardo de um segmento

retardado em uma solução válida, o segmento deve ser forçado a preceder ou preemptar outro segmento escalonado antes dele. Para estabelecer os segmentos que podem ser precedidos ou preemptados pelo segmento mais retardado são definidos os conjuntos de segmentos **Z, G1** e **G2**.

Seja j o segmento mais retardado em uma solução válida. Caso exista mais de um, deixe j ser o segmento que terminou por último entre estes segmentos. Define-se o conjunto de segmentos $Z[j]$ recursivamente como segue:

- $j \in Z[j]$;
- $\forall\ k$:

  Se $\exists\ l,\ l \in Z[j]$:

  $(e[k] = s[l] \wedge (\exists\ l',\ l' \in Z[j]:$

  $r'[l'] < e[k])) \vee$

  $(s[l] \leq e[k] < e[j])$

  então

  $k \in Z[j]$

Em qualquer escalonamento que corresponde à uma solução válida temos que:

- $Z[j]$ é o conjunto de segmentos que precedem e incluem j em um período de utilização contínua do processador. Através do diagrama de Gantt de uma solução válida, pode-se observar que existe uma lacuna(tempo ocioso de processador) antes do conjunto $Z[j]$. O motivo para não considerar os segmentos antes da lacuna é que estes segmentos não podem melhorar o escalonamento do segmento mais retardado, porque a lacuna é criada devido aos tempos de pronto dos segmentos de $Z[j]$. Os segmentos escalonados após o segmento mais retardado também são excluídos do conjunto pelo mesmo motivo;
- $e[j]$ é o tempo de término de execução mais cedo possível para todos os segmentos do conjunto $Z[j]$. Se alguma outra ordem de execução para os segmentos de $Z[j]$ é determinada, o último segmento nesta nova ordem não pode terminar sua execução antes de $e[j]$;
- qualquer solução válida não ótima pode ser

melhorada somente escalonando algum segmento $k \in \mathbf{Z}[j]$, $k \neq j$, tal que $da[j] < da[k]$, após o segmento mais retardado j. Isto é devido ao fato de que o tempo de término de execução mais cedo possível para todos os segmentos de $\mathbf{Z}[j]$ é $e[j]$. Se um segmento de $\mathbf{Z}[j]$ que possui um "deadline" menor ou igual ao do segmento j fosse escalonado após o segmento j, o seu retardo seria, na melhor das hipóteses, igual ao de j.

Para restringir ainda mais o número de segmentos que podem reduzir o retardo do segmento mais retardado em uma solução válida, dois subconjuntos de $\mathbf{Z}[j]$, **G1** e **G2**, são definidos. Estes conjuntos contém apenas os segmentos que garantem alguma redução do retardo do segmento mais retardado.

O conjunto dos segmentos que, se escalonados após o segmento j, podem reduzir o retardo do escalonamento é dado por:

$$\begin{aligned}\mathbf{G1} = \{i \mid\ & i \in \mathbf{Z}[j] \wedge da[j] < da[i] \wedge \\ & (i\ \mathbf{EX}\ j) \wedge \\ & \neg(i\ \mathbf{PC}\ j) \wedge \\ & \neg(i\ \mathbf{PM}\ j)\}.\end{aligned}$$

O conjunto dos segmentos que, se preemptados pelo segmento j, podem reduzir o retardo do escalonamento é dado por:

$$\begin{aligned}\mathbf{G2} = \{i \mid\ & i \in \mathbf{Z}[j] \wedge da[j] < da[i] \wedge \\ & \neg(i\ \mathbf{EX}\ j) \wedge \\ & \neg(i\ \mathbf{PC}\ j) \wedge \\ & \neg(i\ \mathbf{PM}\ j) \wedge \\ & \neg(\exists\ l:\ ((\exists\ k,\ t:\ 0 \leq k \leq (c[l] - 1), \\ & \qquad 0 \leq t \leq \infty:\ s[i] \leq \Pi(l,\ k) \leq e[j]) \wedge \\ & \qquad /^*\ \text{segmento } l \textit{ executa} \text{ entre } i \text{ e } j\ ^*/ \\ & \qquad ((i\ \mathbf{PC}\ l) \vee (i\ \mathbf{PM}\ l)))))\}\end{aligned}$$

Os elementos dos conjuntos **G1** e **G2** são chamados *segmentos candidatos*, e representam todas as possibilidades de se melhorar uma solução válida. Um nó filho é criado para cada segmento candidato. Quando um nó filho é criado, um conjunto de restrições de sincronização para o nó é determinado. Este conjunto consiste de

todas as restrições de sincronização de seu nó pai, acrescido das restrições necessárias para forçar o segmento mais retardado do nó pai a preceder ou preemptar o segmento candidato.

Em cada nó filho, as restrições de sincronização adicionais às de seu nó pai são definidas da seguinte forma:

- se o segmento candidato *i* pertence ao conjunto **G1**, então é adicionado ao conjunto de restrições do nó uma restrição do tipo (*j* **PC** *i*), onde *j* é o segmento mais retardado do nó pai;
- caso o segmento candidato *i* pertença ao conjunto **G2**, então o segmento *j* pode preemptar o segmento candidato. O objetivo neste caso é forçar o escalonamento de pelo menos uma parte do segmento candidato após o segmento *j*. Para isto, é adicionada uma restrição do tipo (*j* **PM** *i*) ao conjunto de restrições do nó filho. Contudo, caso existam segmentos entre o segmento candidato *i* e o segmento mais retardado *j*, restrições adicionais entre cada um destes segmentos e o segmento candidato são requeridas para permitir que o segmento candidato seja preemptado pelo segmento mais retardado. Para todos os segmentos *l* tal que (*i* **EX** *l*) e o segmento *l* executa entre *i* e *j*, é adicionado uma restrição (*l* **PC** *i*), e para todos os segmentos *q* tal que ¬(*i* **EX** *q*) e o segmento *q executa* entre *i* e *j* é adicionado uma restrição (*q* **PM** *i*).

Após os conjuntos de restrições de sincronização de todos os nós filhos terem sido determinados, pode-se gerar uma solução válida para cada nó filho aplicando o algoritmo da figura 3.3. A seguir, verifica-se se alguma das soluções válidas obtidas é ótima ou praticável. Caso não exista nenhuma solução ótima ou praticável entre as soluções válidas geradas, procede-se a criação de novos nós sucessores.

3.4 - BUSCA de uma SOLUÇÃO ÓTIMA ou PRATICÁVEL

3.4.1 - MÉTODOS "BRANCH and BOUND"

O problema de escalonar um conjunto de processos sujeitos a restrições de tempo de pronto, "deadline" e arbitrárias relações de precedência e exclusão mútua é conhecido ser "NP-hard", o que efetivamente exclui a possibilidade de existência de algum algoritmo de complexidade polinomial para solucionar o problema [21], [22]. Para se alcançar o objetivo de encontrar um escalonamento ótimo ou praticável sempre que algum existir, faz-se necessário usar um *método enumerativo*, o qual, simplesmente, após listar ou enumerar todas as possíveis soluções, elimina os escalonamentos não ótimos ou não válidos da lista.

Um método enumerativo pode ser do tipo *explícito* ou *implícito*. Na enumeração explícita, procede-se uma enumeração completa de todas as possíveis soluções e a seguir escolhe-se a melhor segundo algum critério. As técnicas de enumeração implícita, apesar de também considerarem todas as soluções, não o fazem de modo explícito, pois exploram de modo inteligente o conjunto de soluções na tentativa de reduzir o esforço de busca.

Para buscar por uma solução válida praticável ou ótima, o algoritmo de Xu e Parnas[21] usa uma técnica de enumeração implícita "branch and bound". No processo de busca, o método "branch and bound" gera uma árvore de busca que tem em seu nó raiz uma solução válida inicial. Cada nó da árvore de busca corresponde à uma solução válida completa, mas não necessariamente praticável. Cada nó filho representa uma maneira diferente de reduzir o retardo do segmento mais retardado da solução válida de seu nó pai. Para reduzir o esforço de busca, um procedimento de *"bounding"* é usado para descartar nós filhos que não podem levar à uma solução melhor do que a melhor solução encontrada até o momento na busca. O procedimento de "bounding" é realizado calculando um *"lowerbound"* do retardo do escalonamento para cada nó filho. O valor de "lowerbound" de um nó é

uma estimativa para o menor valor de retardo de escalonamento possível que pode ser obtido em alguma solução válida de seus nós descendentes. No processo de busca, comparando-se o valor do retardo da melhor solução válida obtida até o momento com o valor do "lowerbound" de um nó, determina-se a necessidade de continuar ou não explorando o ramo da árvore. Virtualmente, métodos "branch and bound" podem ter que explorar completamente todos os nós da árvore de busca, sendo tão ruins quanto uma enumeração completa mas, em geral, apresentam um desempenho bem melhor[25].

A eficiência dos métodos "branch and bound" tem uma relação direta com a qualidade do "lowerbound". Um "lowerbound" é considerado bom quando ele não é muito menor do que o menor valor real da medida de desempenho, eliminando uma grande quantidade de nós em níveis de profundidade da árvore de busca mais altos, reduzindo substancialmente o esforço de busca. Contudo, deve haver um compromisso entre a qualidade do "lowerbound" e o esforço para sua computação.

3.4.2 - CÁLCULO do "LOWERBOUND"

Usando o fato de que $e[j]$ é o tempo de término de execução mais cedo possível para todos os segmentos do conjunto $\mathbf{Z}[j]$, onde j é o segmento mais retardado em uma solução válida, um "lowerbound" do retardo de qualquer solução válida pode ser calculado de acordo com o procedimento da figura 3.4[21].

Analisando-se o procedimento da figura 3.4 pode-se observar que se o conjunto $\mathbf{K}[j]$ é vazio, então o retardo do segmento j não pode ser reduzido. Isto é devido ao fato de que se algum outro segmento $k \in \mathbf{Z}[j]$, onde $da[k] \leq da[j]$ fosse escalonado após o segmento j, então o seu retardo seria, na melhor das hipóteses, igual ao de j.

Se (k **EX** j) e $s[k] < min\{r'[l] \mid s[k] < s[l] \leq s[j]\}$, escalonar k após j implicaria em uma lacuna("gap") no novo escalonamento iniciando em $s[k]$ e terminando em $min\{r'[l] \mid s[k] <$

$s[l] \leq s[j]\}$ e o retardo do novo escalonamento seria no mínimo $e[j]$ - $da[k]$ mais o tamanho da lacuna. LB2[j] é um "lowerbound" trivial do retardo de qualquer segmento j.

```
K[j] := {k | k ∈ Z[j] ∧ k ≠ j ∧ da[j] < da[k] ∧
          ¬(k PC j) ∧
          ¬(k PM j)}
Se K[j] = ø
então
  LB[j] := e[j] - da[j]
senão
  LB[j] := e[j] + min{GAP[k, j] - da[k] | k ∈ K[j]}

LB1[j] := min{LB[j], e[j] - da[j]}
LB2[j] := r'[j] + c[j] - da[j]
lowerbound := max{LB1[j], LB2[j]}

GAP[k, j]:
Se ¬(k EX j)
então
  GAP[k, j] := 0
senão
  GAP[k,j] := max{0, (min{r'[l] | l ∈ Z[j] ∧ k ≠ l ∧
                              s[k] < s[l] ≤ s[j] ∧
                              ¬(k PC l)}) - s[k]}
```

Figura 3.4 - Cálculo do "Lowerbound" do Retardo de uma Solução Válida

3.4.3 - OBTENÇÃO de uma SOLUÇÃO VÁLIDA PRATICÁVEL ou ÓTIMA

Os principais passos do algoritmo "off-line" de Xu e Parnas[21] são:

- Calcule uma solução válida inicial e seu correspondente "lowerbound". Encontre o segmento mais retardado *j* e seu retardo. Se o retardo do escalonamento é igual ao "lowerbound", então **pare** - a solução válida é **ótima**. Caso contrário, chame o nó raiz de nó pai.

- Para a solução válida do nó pai, encontre os conjuntos **Z**, **G1** e **G2** e crie **|G1|** + **|G2|** novos nós filhos. Deixe cada nó filho herdar todas as restrições de sincronização de seu nó pai. Em cada nó filho, defina as restrições de sincronização adicionais às de seu nó pai, necessárias para forçar o escalonamento mais cedo do segmento mais retardado da solução válida do nó pai. Para cada nó filho, calcule uma solução válida, o "lowerbound" e encontre o segmento mais retardado e seu retardo.

- Se alguma das soluções válidas dos nós filhos é **ótima** ou **praticável**, então **pare**.

- Chame o nó não expandido com menor "lowerbound" de nó pai. Caso exista mais de um, deixe o nó pai ser aquele com mínimo retardo entre estes nós. Retorne ao segundo passo do algoritmo

O processo de criação de novos nós continua até que uma solução válida praticável é encontrada, caso isto seja considerado suficiente, ou até que não exista nenhum nó não expandido que tenha um "lowerbound" menor do que o menor retardo de todas as soluções válidas encontradas até o momento. No último caso, a solução válida que possui o menor retardo de escalonamento é uma solução ótima.

No apêndice B tem-se uma versão do algoritmo em pseudo-código.

3.5 - CONSIDERAÇÕES FINAIS

Neste capítulo, apresentou-se um algoritmo de escalonamento "off-line" ótimo, disponível na literatura[21]. Devido à complexidade dos problemas de escalonamento e da escassez de tempo disponível para um escalonador "on-line" computar escalonamentos dinamicamente, o uso de escalonamento "off-line" torna-se essencial para garantir que as rígidas restrições temporais de processos sejam cumpridas em STRC complexos.

Muitos pregam que se um problema de escalonamento é "NP-hard"[22], então deve-se usar um algoritmo heurístico ao invés de um algoritmo ótimo para solucionar o problema. Contudo, optou-se por usar o algoritmo ótimo de Xu e Parnas[21] devido às seguintes razões:

- embora seja possível construir instâncias do problema nas quais o algoritmo, para encontrar uma solução praticável ou ótima, gaste uma quantidade de tempo de computação que é exponencialmente relacionado ao tamanho da instância, estas instâncias são atípicas e dificilmente são encontradas na prática[23];
- em cada estágio intermediário do algoritmo, uma completa solução válida é construída. Caso o algoritmo seja terminado prematuramente, tem-se uma solução válida que é no mínimo tão boa quanto a solução gerada na raiz da árvore de busca;
- quando escalonamentos são computados "off-line", o tempo gasto não é o fator mais importante. De maior interesse é a habilidade do algoritmo de encontrar um escalonamento sempre que algum existir ou determinar que um escalonamento praticável não existe. No último caso, o projetista tem uma clara indicação de quais parâmetros necessitam ser alterados a fim de se obter um escalonamento praticável, bastando para isso centrar sua atenção no segmento mais retardado e no conjunto de segmentos que precedem e incluem este em um período de utilização contínua do processador.

Deve-se fazer uma distinção entre algoritmos de escalonamento ótimos e heurísticos. O algoritmo de Xu e Parnas[21] é ótimo porque é capaz de encontrar um escalonamento praticável sempre

que algum existir, embora faça uso de uma técnica de busca heurística para tentar reduzir o esforço de busca("branch and bound"). Em contrapartida, algoritmos heurísticos podem falhar em encontrar um escalonamento praticável mesmo quando um existir.

Em STRC, a maior parte da computação pode ser confinada em processos periódicos[24], podendo-se determinar os tempos de pronto das diversas instâncias de cada processo periódico. Os processos periódicos, possuindo tempos de pronto determinísticos, podem ser escalonados "off-line"(inclusive para processos com "deadline" < período) aplicando o algoritmo apresentado neste capítulo para todas as instâncias de processos periódicos que ocorrem dentro de um período de tempo igual ao mínimo múltiplo comum dos períodos. O resultado obtido é armazenado em tabela e usado, em tempo de execução, por um sistema operacional ou núcleo de tempo real.

No próximo capítulo, propõe-se um procedimento para o atendimento dos processos esporádicos.

CAPÍTULO 4

ATENDIMENTO "ON-LINE" DE PROCESSOS ESPORÁDICOS

ATENDIMENTO "ON-LINE" DE PROCESSOS ESPORÁDICOS

4.1 - INTRODUÇÃO

O escalonamento em **STRC** não pode ser completamente determinado "off-line" devido à existência de processos esporádicos (processos que podem ser invocados a qualquer instante, não possuindo tempo de chegada determinístico), necessários para tratar eventos randômicos tais como falhas, requisições do operador, etc. Para estes processos faz-se necessário uma abordagem "on-line", visto que o algoritmo apresentado no capítulo anterior não pode ser usado "on-line" por possuir complexidade exponencial.

O procedimento proposto neste trabalho para o atendimento "on-line" de processos esporádicos é constituído, basicamente, de três etapas[27]:

- extração de informações do escalonamento obtido "off-line" para os processos periódicos;
- teste de aceitação "on-line" executado a cada invocação de processo esporádico;
- política para escalonar o processo esporádico, caso este seja aceito, sem comprometer o escalonamento obtido "off-line" para os processos periódicos.

O teste de aceitação, baseado nas informações extraídas do escalonamento obtido "off-line", verifica se um processo esporádico cumprirá ou não seu "deadline", permitindo, no caso de não ser possível, tomar ações preventivas particulares de cada aplicação, as quais serão discutidas mais adiante. A decisão de aceitar ou não um processo esporádico deve ser tomada o mais rápido possível para não comprometer os requisitos temporais da aplicação. Uma vez aceito, o processo deve ser escalonado sem comprometer o escalonamento gerado em modo "off-line" para os processos periódicos.

4.2 - EXTRAÇÃO de INFORMAÇÕES do ESCALONAMENTO "OFF-LINE"

Podemos extrair uma série de informações úteis para o atendimento "on-line" de processos esporádicos do escalonamento obtido para os processos periódicos. Estas informações, também computadas "off-line", quando organizadas em estruturas de dados adequadas, permitem um eficiente atendimento "on-line" para os processos esporádicos.

Consideremos os processos periódicos A e B dados na figura 4.1. Para uma instância do processo A, por exemplo, o "deadline" absoluto *da* ocorrerá 6 unidades de tempo após sua invocação. Os processos compartilham um recurso 1 qualquer e as regiões dos processos que acessam este recurso aparecem indicadas. O escalonamento destes, feito "off-line" através do algoritmo apresentado no capítulo anterior, é mostrado por meio de um diagrama de Gantt. Este exemplo servirá para ilustrar vários conceitos introduzidos a seguir.

Processos Periódicos	c	p	d
A	4	10	6
B	8	20	20

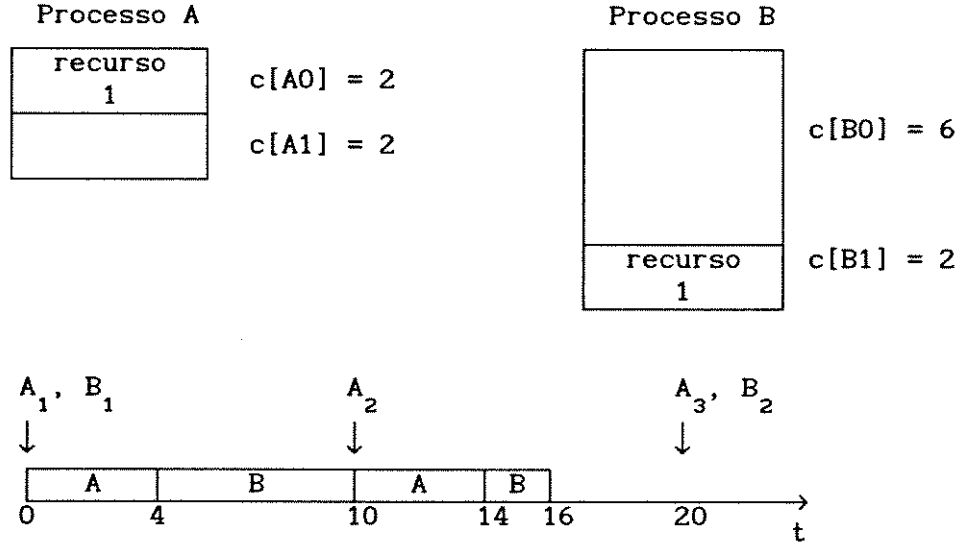


Figura 4.1 - Escalonamento "Off-line" de Processos

O escalonamento de processos periódicos apresenta uma propriedade fundamental: o caráter cíclico[28]. Esta propriedade permite estabelecer um limite de tempo a partir do qual a sequência de execução dos processos periódicos repete-se. O valor deste limite de tempo, aqui denominado *tamanho da janela cíclica*, será denotado por w. Caso a primeira instância de todos os processos periódicos P_1, P_2, ..., P_n tenham tempo de chegada igual a 0, ou seja, $a[P_{i,1}] = 0$, $i = 1, 2, \ldots, n$, o tamanho da janela cíclica é dado por: $w = m.m.c.\{p_1, p_2, \ldots, p_n\}$, onde p_i é o período do processo P_i. O intervalo $[(j - 1).w, j.w)$, $j \geq 1$, será chamado *j-ésima janela cíclica*. Como as sequências de execução dos processos periódicos nos intervalos $[0, w)$, $[w, 2w)$, $[2w, 3w)$, ... são idênticas, o escalonamento obtido para todas as instâncias de processos periódicos que ocorrem no intervalo $[0, w)$ nos possibilita conhecer a ocupação do processador por processos periódicos em qualquer instante $t \in [0, \infty)$. No exemplo da figura 4.1, temos que $w = m.m.c.\{10, 20\}$, ou seja, $w = 20$.

Chama-se *bloco* cada sequência contínua de ocupação do processador por um mesmo processo periódico. Na figura 4.1 temos 4 blocos: bloco b_1, que inicia em $t = 0$ e termina em $t = 4$; bloco b_2, que inicia em 4 e termina em 10; bloco b_3, que inicia em 10 e termina em 14, e o bloco b_4, que inicia em 14 e termina em 16.

No escalonamento da figura 4.1 a primeira instância do processo A deu origem ao bloco b_1, a segunda ao bloco b_3, e a instância do processo B deu origens aos blocos b_2 e b_4. O último bloco de cada instância de processo periódico na primeira janela cíclica será chamado *bloco terminal*. No exemplo, apenas b_2 é um *bloco não-terminal*.

Seja $e[X_i]$ o instante de término de execução da i-ésima instância de um processo X qualquer. O *retardo* ("lateness") de um processo X_i, $l[X_i]$, é definido como a diferença entre o seu instante de término e o seu "deadline" absoluto, ou seja:

$$l[X_i] = e[X_i] - da[X_i]$$

Na figura 4.1, tem-se:

$$l[A_1] = 4 - 6 = -2;$$
$$l[A_2] = 14 - 16 = -2;$$
$$l[B_1] = 16 - 20 = -4.$$

Observa-se que se $l[X_i] \leq 0$, o processo X_i cumpre o seu "deadline". Se b_k é o bloco terminal de um processo X_i, definimos o retardo de b_k como:

$$l[b_k] = l[X_i]$$

Caso exista um escalonamento praticável para os processos periódicos (todos os "deadlines" são cumpridos), podemos calcular o *atraso máximo* de cada bloco, que é o máximo valor que cada bloco no escalonamento obtido "off-line" inicialmente pode atrasar o seu término sem comprometer os "deadlines" dos processos periódicos. O objetivo é poder atrasar os blocos para atender requisições de processos esporádicos. A figura 4.2 apresenta um algoritmo para o cálculo dos atrasos máximos dos blocos b_k, $amax[b_k]$, para $1 \leq k \leq n$, onde n neste caso representa o número de blocos executados na primeira janela cíclica.

```
amax[b_n] := |l[b_n]|
Para k := (n - 1) até 1 faça
início
  amax[b_k] := início[b_(k + 1)] - fim[b_k] +
               + amax[b_(k + 1)]
  Se (b_k é terminal)
  então
    amax[b_k] := min { |l[b_k]|, amax[b_k] }
fim
```

Figura 4.2 - Cálculo dos Atrasos Máximos dos Blocos

Aplicando o algoritmo da figura 4.2 aos blocos da figura 4.1 obtem-se:

$$amax[b_4] = 4;$$
$$amax[b_3] = 2;$$
$$amax[b_2] = 2;$$
$$amax[b_1] = 2.$$

Uma vez calculado os atrasos máximos para todos os blocos que executam em [0, w), podemos calcular o tempo máximo que o processador pode ser utilizado para executar processos esporádicos em um intervalo [t_1, t_2] qualquer, t_1 e t_2 pertencendo à primeira janela cíclica, sem comprometer o escalonamento gerado "off-line" para os processos periódicos. Este tempo máximo, dado por *omega*(t_1, t_2), é o tempo ocioso do processador no intervalo [t_1, t_2] do escalonamento "off-line" original, acrescido do tempo obtido com o atraso máximo de todos os blocos que executam em [t_1, t_2]:

$$omega(t_1, t_2) = t_2 - t_1 - \sum_{k=i}^{j} (c'[b_k] - min\{c'[b_k], max\{0, (fim'[b_k] + amax[b_k] - t_2)\}\})$$

onde $c'[b_k]$ e $fim'[b_k]$ são, respectivamente, o número de unidades de tempo do bloco b_k que executa no intervalo [t_1, t_2], e o instante de execução da última unidade do bloco b_k no intervalo [t_1, t_2]; i e j são, respectivamente, os índices do primeiro e do último bloco que executam alguma unidade de tempo no intervalo [t_1, t_2] do escalonamento "off-line" original. Como exemplo, calculemos *omega*(8, 15) em relação ao escalonamento da figura 4.1:

$$omega(8, 15) = 15 - 8 - ((2 - min\{2, max\{0, 10 + 2 - 15\}\}) + (4 - min\{4, max\{0, 14 + 2 - 15\}\}) + (1 - min\{1, max\{0, 15 + 4 - 15\}\}))$$
$$omega(8, 15) = 7 - (2 + 3 + 0) = 2.$$

Este resultado significa que um processo esporádico com tempo de chegada igual a 8 e "deadline" igual a 15 pode receber, no máximo, 2 unidades de tempo de processador. Se o seu tempo de execução for superior a este valor, o seu "deadline" não será cumprido, devendo ser recusado pelo teste de aceitação.

4.3 - TESTE de ACEITAÇÃO "ON-LINE" para PROCESSOS ESPORÁDICOS

O *teste de aceitação* permite determinar, no instante de chegada de um processo esporádico, se este cumprirá ou não seu "deadline". Processos esporádicos não possuem relação de precedência com processos periódicos, pois isto implicaria em uma completa ausência de previsibilidade do sistema. Caso os processos esporádicos não possuam restrições de exclusão mútua, a figura 4.3 apresenta um algoritmo do teste de aceitação para um processo esporádico $s = (a, c, d)$, onde a é o tempo de chegada, c o tempo de execução e d o "deadline" relativo em relação ao tempo de chegada, com $d > 0$.

O teste de aceitação assume que, quando um processo esporádico requer execução, todos os processos esporádicos previamente aceitos completaram suas execuções. No algoritmo da figura 4.3, *ax* é o tamanho do intervalo entre o início da janela que contém *a* e *a*; *dx* é o tamanho do intervalo entre o início da janela que contém *da* e *da*; *delta* é a quantidade total de tempo ocioso do processador na primeira janela cíclica do escalonamento "off-line" original (no exemplo da figura 4.1, temos *delta* = 4); e *phi*, é o tempo máximo de processador que o processo esporádico pode receber antes do "deadline". Os operadores **div** e **mod** fornecem, respectivamente, o resultado inteiro da divisão entre dois inteiros e o resto da divisão entre dois inteiros.

Quando o tempo de chegada de um processo esporádico ocorre durante a execução de um bloco que na janela cíclica corrente já tenha sido atrasado (para atender uma outra requisição de processo esporádico), é necessário calcular um fator de correção (*fc*), pois a função *omega* é computada considerando os atrasos máximos dos blocos. A seguir, tem-se o cálculo do fator de correção e a versão 1 do teste

de aceitação completa, incluindo este cálculo.

```
fc := 0
da := a + d
ax := a mod w
dx := da mod w
aw := (a div w) + 1
dw := (da div w) + 1
Se (aw = dw)
então   /* a e da estão na mesma janela cíclica */
  phi := omega(ax, dx)
senão   /* a e da estão em janelas diferentes */
  phi := omega(ax, w) + (dw - aw - 1) * delta +
         omega(0, dx)
Se ("processador não ocioso" e
    "bloco em execução já foi atrasado"
então
  "calcular fator de correção fc"
phi := phi - fc
Se (c ≤ phi)
então
  ("aceita-se o processo esporádico s")
senão
  ("não se aceita o processo esporádico s")
```

Figura 4.3 - Teste de Aceitação - Versão 1 Simplificada

4.3.1 - CÁLCULO do FATOR de CORREÇÃO

A função *omega*(t_1, t_2) é calculada aplicando o atraso máximo a todos os blocos que executam alguma unidade de tempo no intervalo [t_1, t_2] do escalonamento "off-line" original. Considere um processo esporádico s = (a, c, d). Caso o seu tempo de chegada a ocorra durante a execução de um bloco b_{pb}, que na janela cíclica

corrente já tenha sido atrasado para atender outras requisições de processos esporádicos, o processo *s* terá acesso a um tempo de processador antes de seu "deadline" menor, pois uma parte (ou todo) do atraso máximo do bloco b_{pb} já foi utilizado por outros processos esporádicos. Nesse caso, surge a necessidade de um fator de correção *fc* para o valor *phi*.

Todos os blocos possuem um indicador de *atraso permitido* (*ap*) os quais, em todo início de janela cíclica, são iniciados com os seus respectivos valores de atraso máximo, previamente calculados. Sempre que um bloco for atrasado, o seu indicador de atraso permitido é atualizado. Toda vez que um processo esporádico *s* = (*a*, *c*, *d*) chega durante a execução de um bloco b_{pb}, necessita-se do valor de *omega*(*ax*, *dx*) ou *omega*(*ax*, *w*) para o teste de aceitação, dependendo dos valores de *a* e *da* estarem ou não na mesma janela cíclica. Seja *omega*(*ax*, t_2), com t_2 podendo assumir os valores de *dx* ou *w*. Temos então duas situações:

- $t_2 \geq (ax + amax[pb])$: neste caso, o bloco b_{pb} contribui para *omega*(*ax*, t_2) com o valor *amax*[*pb*];
- $t_2 < (ax + amax[pb])$: o bloco b_{pb} contribui para *omega*(*ax*, t_2) com o valor $(t_2 - ax)$.

A figura 4.4 apresenta um algoritmo do teste de aceitação para um processo esporádico *s* = (*a*, *c*, *d*), incorporando o cálculo do fator de correção discutido acima.

```
fc := 0
da := a + d

ax := a mod w
dx := da mod w
aw := (a div w) + 1
dw := (da div w) + 1
Se (aw = dw)
então   /* a e da estão na mesma janela cíclica */
início
  phi := omega(ax, dx)
  t2 := dx
fim
senão   /* a e da estão em janelas diferentes */
início
  phi := omega(ax, w) + (dw - aw - 1) * delta +
           omega(0, dx)
  t2 := w
fim
Se ("processador não ocioso")
então
início
  pb := "índice do bloco em execução"
  Se (amax[pb]  ≠ ap[pb])
  então  /* Calcular fator de correção */
    Se (t2 ≥ ax + amax[pb])
    então
      fc := amax[pb] - ap[pb]
    senão
      Se (ap[pb] < t2 - ax)
      então
        fc := t2 - ax - ap[pb]
fim
.
.
.
```

```
phi := phi - fc
Se (c ≤ phi)
então
  ("aceita-se o processo esporádico s")
senão
  ("não se aceita o processo esporádico s")
```

Figura 4.4 - Teste de Aceitação - Versão 1 Completa

4.4 - ESCALONAMENTO de PROCESSOS ESPORÁDICOS

No caso de um processo esporádico ser aceito, isto é, $c \leq phi$, a próxima etapa consistirá no escalonamento deste, alocando para o processo tempo ocioso de processador e o tempo obtido com o atraso dos blocos do escalonamento "off-line" original. Quando um processo esporádico chega e é aceito, tem-se a *garantia* de que será possível executá-lo e cumprir o seu "deadline" sem comprometer os "deadlines" dos processos periódicos previamente garantidos. A estratégia de escalonamento consiste em dar prioridade ao processo esporádico, atrasando o máximo possível os blocos de processos periódicos, ou seja, até o limite de seus indicadores de atraso permitido.

O valor de *phi* é uma caracterização exata do tempo máximo de processador que pode ser destinado a um processo esporádico $s = (a, c, d)$ no intervalo $[a, a + d]$, sem comprometer o escalonamento "off-line". Portanto, em uma situação em que $c > phi$, não existe a possibilidade do processo esporádico cumprir o seu "deadline" sem comprometer os "deadlines" dos processos esporádicos previamente garantidos, devendo ser tomadas ações preventivas particulares de cada aplicação. Estas ações podem variar desde, por exemplo, anunciar o problema no painel do operador e executar tratadores de exceção, até ações mais sofisticadas como ter várias versões funcionais do mesmo processo esporádico para aquela situação e o núcleo escolher a versão cujo tempo de execução c cumpra a

restrição $c \leq phi$.

Para exemplificar, consideremos um processo esporádico s chegando no instante 8, com "deadline" igual a 7 e tempo de execução igual a 2 (s = (8, 2, 7)), no cenário da figura 4.1. Já sabemos que *omega*(8, 15) = 2 e, portanto, o processo esporádico s pode ser aceito. O escalonamento deste é apresentado na figura 4.5. Observe que o escalonamento "off-line" não foi comprometido.

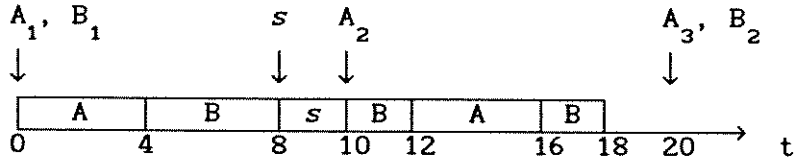


Figura 4.5 - Exemplo de Escalonamento de Processo Esporádico

4.5 - ATENDIMENTO de VÁRIOS PROCESSOS ESPORÁDICOS

A restrição imposta ao modelo de que, quando um processo esporádico requer execução todos os processos esporádicos previamente aceitos completaram suas execuções, pode ser assegurada através de "bufferização". Assim, um processo esporádico que chega antes do término de execução de um processo esporádico previamente aceito tem que aguardar, e o teste de aceitação para este só será executado após o término de execução do processo previamente aceito. Neste momento pode ser muito tarde, pois o "deadline" do processo pode ter expirado e nenhuma ação preventiva poderá ser tomada. A seguir apresenta-se uma extensão do teste de aceitação que relaxa a restrição citada acima, possibilitando que o teste de aceitação seja executado no instante de chegada de qualquer processo esporádico.

Seja s_i o processo esporádico sendo executado e s_{i+1}, s_{i+2}, ..., s_{n-1} processos esporádicos aceitos aguardando execução, que chegaram depois do processo s_i. O teste de aceitação para um processo esporádico s_n que chega no instante a_n ($s_n = (a_n, c_n, d_n)$) é dado na figura 4.6. É apresentado apenas a última estrutura

condicional, que consiste na única diferença em relação ao teste da figura 4.4.

Na figura 4.6, $c_i^{a_n}$ é o tempo de execução restante do processo esporádico s_i no instante a_n e $phi(a_n, a_n + d_n)$ é o tempo máximo de processador que pode ser utilizado para executar processos esporádicos no intervalo $[a_n, a_n + d_n]$, calculado de modo idêntico ao valor *phi* usado no teste de aceitação da figura 4.4.

```
.
.
.
Se (c_i^{a_n} + c_{i+1} + ... + c_{n-1} + c_n ≤ phi(a_n, a_n + d_n))
então
  ("aceita-se o processo esporádico s_n")
senão
  ("não se aceita o processo esporádico s_n")
```

Figura 4.6 - Teste de Aceitação : Atendimento de Vários Processos Esporádicos - Versão 2

Caso o processo s_n seja aceito, ele só poderá iniciar execução após o término de todos os processos esporádicos previamente aceitos, ocupando o processador de acordo com a política discutida na seção 4.4, ou seja, nos instantes em que o processador estiver ocioso ou naqueles instantes em que um bloco de processo periódico puder ser atrasado.

4.6 - ASPECTOS de IMPLEMENTAÇÃO

As decisões tomadas "on-line" pelo núcleo do sistema operacional devem ser suficientemente rápidas para não comprometer os requisitos temporais da aplicação. O teste de aceitação apresentado neste capítulo é simples e eficiente, desde que os valores da função $omega(t_1, t_2)$ se encontrem disponíveis, calculados "off-line". O cálculo de $omega(t_1, t_2)$ não é praticável em modo "on-line" devido à sua alta complexidade. Como todas as informações necessárias para o cálculo de $omega(t_1, t_2)$ estão disponíveis após o escalonamento "off-line" dos processos periódicos, optou-se por calcular, também "off-line", os valores de $omega(t_1, t_2)$ em todo seu domínio, ou seja $0 \leq t_1 \leq (w - 1)$ e $(t_1 + 1) \leq t_2 \leq w$ (assumindo que as preempções dos processos periódicos ocorrem em instantes de tempo inteiros), e armazená-los em uma tabela.

O tipo de estrutura de dados natural para armazenar todos os valores da função $omega(t_1, t_2)$ é a matriz bidimensional. Contudo, como pode ser visto na figura 4.7, uma matriz não possuiria elementos abaixo da diagonal principal, devido ao fato de $omega(t_1, t_2)$ não ser definida para $t_2 \leq t_1$. Optou-se então por linearizar a matriz e armazená-la em vetor, pois caso a matriz fosse armazenada integralmente, causaria um grande desperdício de memória.

$$\begin{bmatrix} omega(0,1) & omega(0,2) & omega(0,3) & \cdots & omega(0,w) \\ & omega(1,2) & omega(1,3) & \cdots & omega(1,w) \\ & & omega(2,3) & \cdots & omega(2,w) \\ & & \vdots & & \\ & omega(w-2,w-1) & & & omega(w-2,w) \\ & & & & omega(w-1,w) \end{bmatrix}$$

Figura 4.7 - Domínio da Função omega(t_1, t_2)

Uma vez definido o modo de linearização da matriz de valores da função *omega*(t_1, t_2) como sendo por linhas, de cima para baixo, da esquerda para direita, faz-se necessário uma função que, dado o par (t_1, t_2), mapeie estes valores no índice do vetor onde será armazenado o valor de *omega*(t_1, t_2). A função

$$i: (t_1, t_2) \longrightarrow \mathbf{N}$$

$$i(t_1, t_2) = \frac{(2w - t_1 + 1)t_1}{2} + t_2 - t_1$$

onde w é o tamanho da janela cíclica, faz isto. Exemplificando, temos:

$i(0, 1) = 1$: *omega*(0, 1) será armazenado na primeira posição do vetor;

$i(w - 1, w) = (w^2 + w)/2$: *omega*($w - 1$, w) será armazenado na $(w^2 + w)/2$ º posição do vetor, ou seja, a última.

A função *omega*(t_1,t_2) pode assumir $(w^2 + w)/2$ valores. Como w é dado pelo mínimo múltiplo comum dos períodos dos processos periódicos, caso estes sejam relativamente primos, pode ser necessário uma grande quantidade de memória para armazenar todos os possíveis valores. Alternativas para tentar reduzir a tabela incluem o ajuste manual dos períodos, caso seja possível, procurando obter valores que sejam múltiplos entre si para reduzir o tamanho de w, e a identificação de valores da função *omega*(t_1, t_2) não utilizados pelo teste de aceitação, usando o fato de se conhecer previamente os "deadlines" relativos dos processos esporádicos, armazenando na tabela apenas os valores úteis. A seguir, discute-se a segunda alternativa.

Suponha que exista no sistema apenas um processo esporádico $s = (a, c, d)$. Analisa-se primeiro o caso em que $d \leq w$:

- seja $ax = a$ **mod** w, tal como nos algoritmos do teste de aceitação. No caso de ax pertencer ao intervalo $[0, w - d]$, bastariam os seguintes valores de *omega*(t_1, t_2) para o cálculo do valor *phi*:

$$0 \leq ax \leq (w - d)$$
$$phi = omega(ax,\ ax + d).$$

Logo, seriam necessários $(w - d + 1)$ valores de $omega(t_1,\ t_2)$, ou entradas na tabela para este caso;

●considerando o caso em que ax pertence ao intervalo $[w - d + 1,\ w - 1]$, bastariam os seguintes valores de $omega(t_1,\ t_2)$ para o cálculo de *phi*:

$$(w - d + 1) \leq ax \leq (w - 1)$$
$$phi = omega(ax,\ w) + omega(0,\ ax + d - w).$$

Logo, como ax neste caso pode assumir $(d - 1)$ valores, serão necessários $2(d - 1)$ valores de $omega(t_1,\ t_2)$, ou entradas na tabela. Portanto, no caso em que $d \leq w$, são necessários $(w - d + 1) + 2(d - 1)$, ou seja, $(w + d - 1)$ valores de $omega(t_1,\ t_2)$ para o cálculo de *phi*.

Analisando agora o caso em que $d > w$ observa-se que para o cálculo de *phi* são necessários os seguintes valores de $omega(t_1,\ t_2)$:

$$0 \leq ax \leq (w - 1)$$
$$phi = omega(ax,\ w) + (dw - aw - 1)delta + omega(0,\ (ax + d) \bmod w).$$

Logo, como ax neste caso pode assumir w valores, serão necessários $2w$ valores de $omega(t_1,\ t_2)$ ou entradas na tabela.

Resumindo, temos:

- para $d \leq w$: são necessários $(w + d - 1)$ entradas na tabela;
- para $d > w$: são necessários $2w$ entradas na tabela.

Portanto, neste caso, o número de entradas é uma função linear de w.

Os resultados acima foram obtidos considerando um único processo esporádico $s = (a,\ c,\ d)$ no sistema e servem para demonstrar a possibilidade de reduzir a um número bem abaixo de $(w^2 + w)/2$ a quantidade de entradas na tabela $omega(t_1,\ t_2)$. Contudo, em uma

aplicação real, existem vários processos esporádicos e uma alternativa de implementação é construir para cada processo esporádico uma tabela contendo valores *phi*, ao longo de toda janela cíclica. As tabelas *phi* são computadas "off-line" considerando o atraso máximo de todos os blocos. Quando um processo esporádico chega, basta recuperar na sua tabela *phi* o valor respectivo ao seu tempo de chegada e calcular o fator de correção, caso seja necessário. Logo, cada tabela *phi* possui w entradas, pois $0 \leq ax \leq (w - 1)$. Caso existam n processos esporádicos no sistema serão necessárias $n.w$ entradas em tabelas. Fazendo

$$n.w = \frac{w^2 + w}{2} \quad \Rightarrow \quad n = \frac{w + 1}{2}$$

chega-se à conclusão que para um número de processos esporádicos menor ou igual a $(w + 1)/2$ é vantajoso em termos de memória e velocidade de processamento usar uma tabela *phi* específica para cada processo esporádico, ao invés de computar a função $omega(t_1, t_2)$ em todo seu domínio.

4.7 - CONSIDERAÇÕES FINAIS

Chetto e Chetto[29] apresentaram um procedimento para o atendimento "on-line" de processos esporádicos em sistemas monoprocessadores, mas apenas para o caso onde os processos (periódicos e esporádicos) são independentes, ou seja, sem restrições de precedência e exclusão mútua. Em [29] o escalonamento de processos periódicos e esporádicos é feito através do algoritmo "earliest deadline".

Na estratégia de escalonamento proposta, usou-se um algoritmo de escalonamento "off-line" ótimo que permite escalonar processos periódicos sujeitos à restrições de tempo de pronto, "deadline", relações de precedência e exclusão mútua. No atendimento "on-line" de processos esporádicos utiliza-se tabelas computadas "off-line" que permitem determinar com eficiência o tempo máximo de

processador que pode ser destinado ao atendimento de processos esporádicos entre dois instantes de tempo quaisquer, sem comprometer os processos periódicos previamente garantidos.

O teste de aceitação "on-line" proposto relaxa a restrição de que, quando um processo esporádico requer execução, todos os processos esporádicos previamente aceitos completaram suas execuções. Esta restrição é assumida no trabalho de Chetto e Chetto[29].

Em sistemas monoprocessados, no caso do teste de aceitação para um processo esporádico falhar, pode-se usar o esquema de múltiplas versões funcionais para um mesmo processo esporádico, com o núcleo escolhendo a versão de maior funcionalidade cujo tempo de execução c cumpra a restrição $c \leq phi$.

Em sistemas distribuídos o escalonamento de processos é feito em dois níveis: *local* e *global*. O escalonador local de um nó é responsável pelo escalonamento dos processos periódicos e esporádicos alocados a este. Quando um processo esporádico chega em um nó e o teste de aceitação para este falha, é acionado o escalonador global do nó. O escalonador global interage com os escalonadores globais dos demais nós do sistema, tentando encontrar uma estação remota que possa garantir a execução do processo não aceito na estação local. A estratégia de escalonamento proposta neste trabalho (procedimentos "off-line" e "on-line") pode ser usada em sistemas distribuídos como um escalonador local. O escalonamento global pode ser realizado pelos algoritmos "bidding" e "focussed adressing"[30], [31]. Em [32] é apresentado um algoritmo específico para o escalonamento local em sistemas distribuídos. Este algoritmo, apesar de considerar requisições de recursos adicionais por parte dos processos (dispositivos de I/O, arquivos, etc.), trata apenas processos não-preemptivos e independentes, o que restringe bastante sua aplicação.

A política de atendimento "on-line" de processos esporádicos proposta neste trabalho é independente da política "off-line", podendo vir a ser usada com outros algoritmos de escalonamento "off-line". O algoritmo "off-line" de Xu e Parnas[21] foi utilizado por ser o mais completo disponível na literatura no

momento.

CAPÍTULO 5

UM ESCALONADOR DE PROCESSOS PERIÓDICOS E ESPORÁDICOS

UM ESCALONADOR DE PROCESSOS PERIÓDICOS E ESPORÁDICOS

5.1 - INTRODUÇÃO

Com o objetivo de validar a estratégia proposta para o escalonamento de processos periódicos e esporádicos em **STRC** monoprocessados, foi implementado um sistema chamado *Escalonador de Processos Periódicos e Esporádicos* - **EPPE**, o qual simula a execução de um conjunto de processos, exibindo a sequência em que são escalonados. O **EPPE** roda sobre o sistema operacional DOS e foi totalmente escrito em linguagem C.

A tela inicial do **EPPE** apresenta os 4 tipos de serviços disponíveis:

1 - Escalonamento off-line
2 - Entrada processos esporádicos
3 - Simulação
4 - Fim

O escalonamento de processos periódicos e o cálculo da função $omega(t_1, t_2)$ em todo seu domínio é realizado selecionando-se o serviço *Escalonamento off-line*. Deve ser fornecido o tamanho da janela cíclica, o número de segmentos a ser escalonados, os parâmetros tempo de chegada, tempo de computação e "deadline" absoluto para cada segmento de processo periódico e, caso existam, as relações de precedência e exclusão mútua entre os segmentos. O escalonamento é feito usando-se o algoritmo "off-line" de Xu e Parnas[21].

Ao selecionar-se o serviço *Entrada processos esporádicos* é pedido o número de processos esporádicos a ser atendido. A seguir, procede-se à entrada dos parâmetros tempo de chegada, tempo de computação e "deadline" absoluto para cada processo

esporádico. Os processos esporádicos devem ser fornecidos em ordem crescente de tempo de chegada. O atendimento destes é feito usando-se os algoritmos propostos no capítulo 4 deste trabalho.

Na opção 3, *Simulação*, após ser fornecido o tempo de simulação, é exibido a sequência em que os processos são escalonados. Para se usar o serviço *Simulação* deve ter sido obtido previamente um escalonamento praticável de processos periódicos.

O serviço *Fim* retorna ao sistema operacional.

A seguir serão discutidos alguns resultados obtidos com o escalonamento de processos periódicos e esporádicos.

5.2 - ESCALONAMENTO de PROCESSOS PERIÓDICOS

Exemplo 1: considere os processos periódicos A, B, C e D dados na figura 5.1. Para uma instância do processo A, por exemplo, o "deadline" absoluto *da* ocorrerá 30 unidades de tempo após sua invocação. O processo B possui 2 pontos de sincronismo com o processo A, o que faz com que o processo B seja decomposto em dois segmentos: B0 e B1. O segmento B0 e o processo D acessam um mesmo recurso 1 qualquer. Considere que as primeiras instâncias de todos os processos periódicos possuem tempo de chegada igual a 0. $B0_1$ é o segmento B0 da primeira instância do processo B (B_1), **PC** indica relação de precedência e **EX** relação de exclusão mútua. Note que as relações não são simétricas.

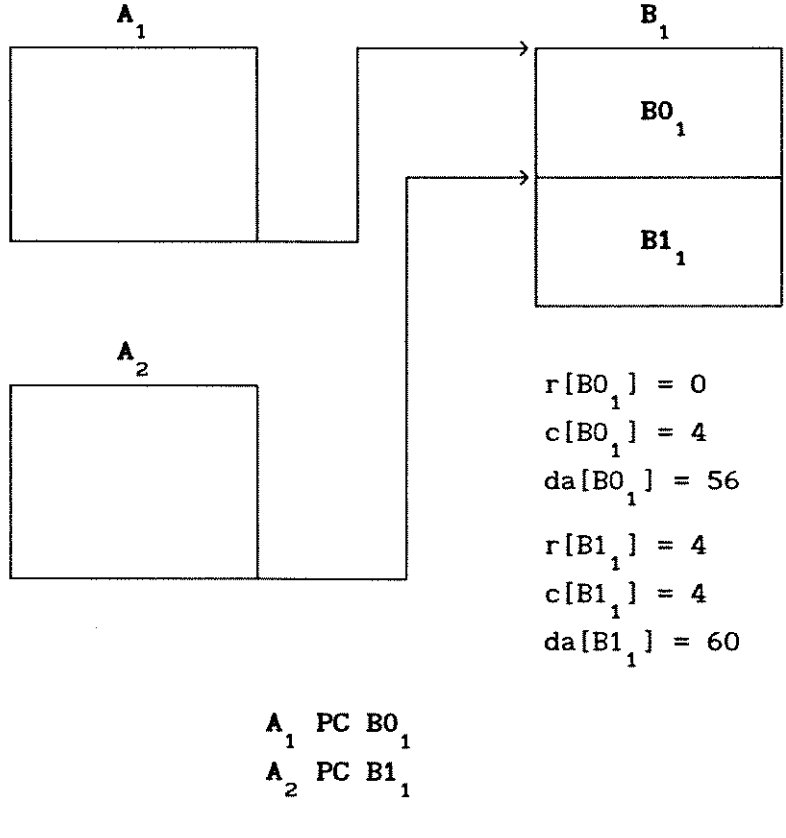
O tamanho da janela cíclica w para este conjunto de processos periódicos é dado por:

$$w = m.m.c.\{30,\ 60,\ 40,\ 60\} \Rightarrow w = 120.$$

Logo, para se obter o conjunto $\mathbf{S(\mathfrak{P})}$, deve-se decompor em segmentos todas as instâncias de processos periódicos que ocorrem no intervalo [0, 120). As relações de precedência e exclusão mútua devem ser definidas entre pares ordenados de segmentos de $\mathbf{S(\mathfrak{P})}$. Estas relações, juntamente com o conjunto $\mathbf{S(\mathfrak{P})}$, constituem a entrada do algoritmo de Xu e Parnas[21].

Processos Periódicos	c	p	d
A	6	30	30
B	8	60	60
C	8	40	40
D	12	60	60

Precedência:



A_1 PC BO_1
A_2 PC $B1_1$

Exclusão:

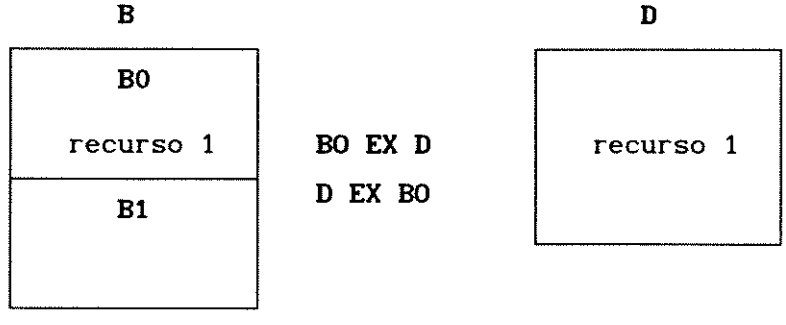


Figura 5.1 - Exemplo 1: Processos Periódicos

Na figura 5.2, apresenta-se a entrada fornecida ao **EPPE** para os processos do exemplo 1 ao selecionar-se a opção *Escalonamento off-line*.

Segmento	Identificador	r	c	da
A_1	1	0	6	30
A_2	2	30	6	60
A_3	3	60	6	90
A_4	4	90	6	120
BO_1	5	0	4	56
$B1_1$	6	4	4	60
BO_2	7	60	4	116
$B1_2$	8	64	4	120
C_1	9	0	8	40
C_2	10	40	8	80
C_3	11	80	8	120
D_1	12	0	12	60
D_2	13	60	12	120

Precedência:

1 **PC** 5, 2 **PC** 6, 3 **PC** 7, 4 **PC** 8

Exclusão:

5 **EX** 12, 7 **EX** 13, 12 **EX** 5, 13 **EX** 7

Figura 5.2 - Exemplo 1: Entrada Fornecida ao EPPE

Após o processamento dos dados de entrada apresentados na figura acima, o **EPPE** gerou um escalonamento ótimo praticável para os processos periódicos, o qual é apresentado na figura 5.3, ao longo da primeira janela cíclica, por meio de um diagrama de Gantt.

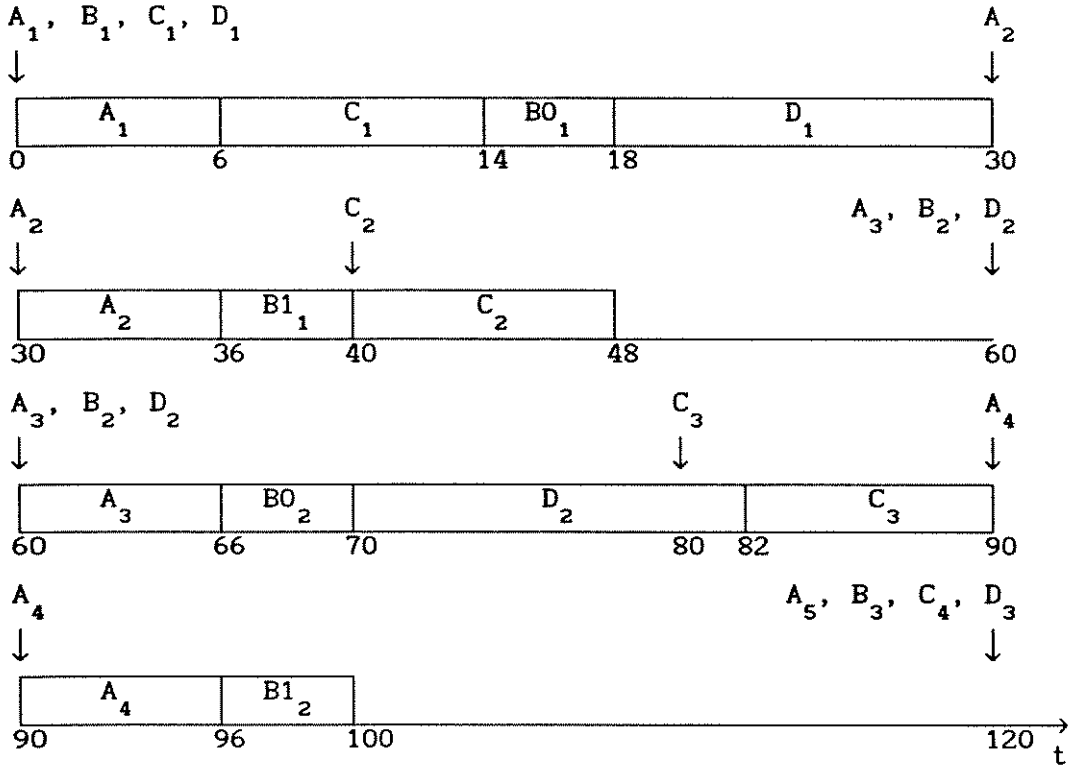


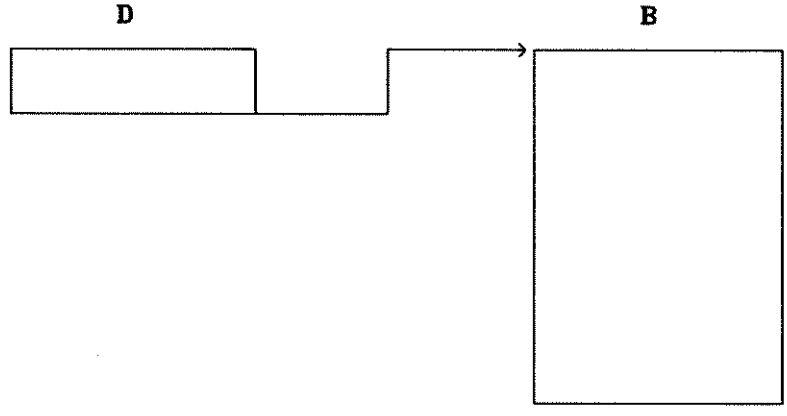
Figura 5.3 - Exemplo 1: Escalonamento de Processos Periódicos

Na obtenção do escalonamento acima, o **EPPE** gerou apenas um nó da árvore de busca, ou seja, a solução válida inicial encontrada foi ótima. O segmento mais retardado é $B1_2$, o qual termina 20 unidades de tempo antes de seu "deadline", ou seja, $l[B1_2] = -20$. Pode-se observar que todas as relações de precedência e exclusão mútua estão satisfeitas no escalonamento obtido, não se fazendo necessário mecanismos de sincronização em tempo de execução.

Exemplo 2: considere os processos periódicos A, B, C e D dados na figura 5.4. Toda instância do processo D deve preceder a execução da instância corrente do processo B. Os processos A, B e C acessam um mesmo recurso 1 qualquer. Assume-se que as primeiras instâncias de todos os processos periódicos possuem tempo de chegada igual a 0.

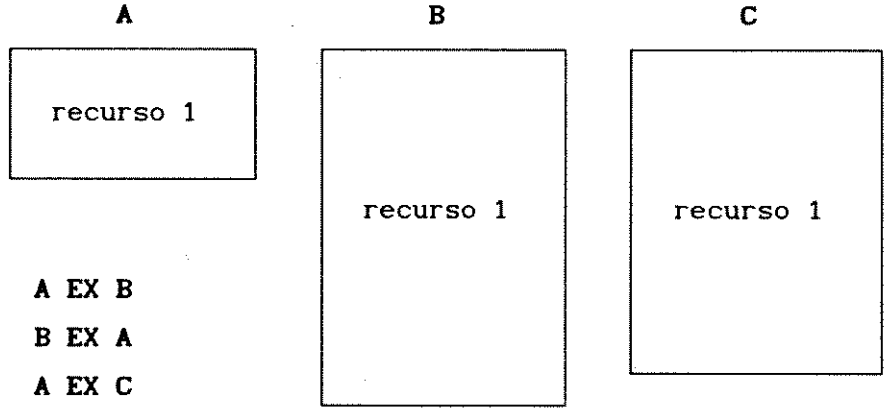
Processos Periódicos	c	p	d
A	4	15	15
B	11	30	25
C	10	45	29
D	2	30	30

Precedência:



D PC B

Exclusão:



A EX B
B EX A
A EX C
C EX A
B EX C
C EX B

Figura 5.4 - Exemplo 2: Processos Periódicos

O tamanho da janela cíclica w para os processos periódicos da figura 5.4 é dado por:

$$w = m.m.c.\{15,\ 30,\ 45,\ 30\} \Rightarrow w = 90.$$

Após decompor-se em segmentos todas as instâncias de processos periódicos que ocorrem no intervalo [0, 90) para se obter o conjunto **S(Ψ)**, e definir as relações de precedência e exclusão mútua entre pares ordenados de **S(Ψ)**, forneceu-se estes dados ao **EPPE**. Para estes dados de entrada, o **EPPE** gerou um escalonamento ótimo não praticável para os processos periódicos, o qual é apresentado na figura 5.5, ao longo da primeira janela cíclica, por meio de um diagrama de Gantt.

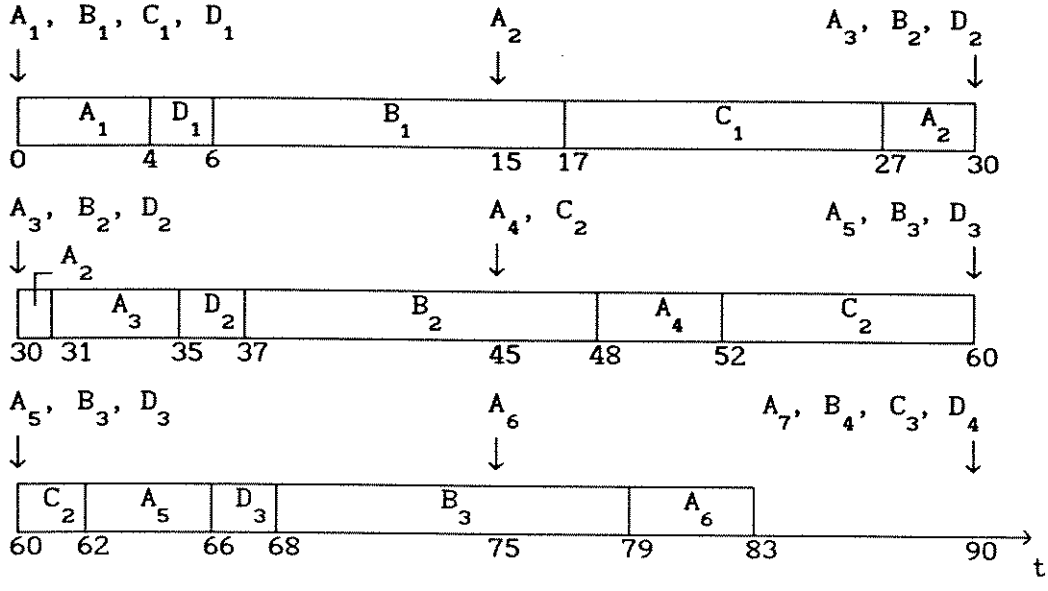


Figura 5.5 - Exemplo 2: Escalonamento de Processos Periódicos

Na obtenção do escalonamento acima, o **EPPE** gerou 4 nós da árvore de busca. O segmento mais retardado é A_2, o qual termina 1 unidade de tempo após seu "deadline", ou seja, $l[A_2] = 1$. Como o algoritmo "off-line" de Xu e Parnas[21] é ótimo, não existe nenhum escalonamento praticável possível para o conjunto de processos periódicos do exemplo 2. Este fato é devido, principalmente, ao alto fator de utilização de processador (**U** = 0,92) e também das complexas relações entre os processos, os quais apresentam regiões críticas bastante extensas.

O serviço do **EPPE** *Escalonamento off-line* foi testado para todos os exemplos apresentados no trabalho de Xu e Parnas[21] e apresentou resultados idênticos aos do artigo. A seguir é apresentado um dos exemplos do artigo.

Exemplo 3: considere um conjunto de segmentos **S(P)**, constituído dos segmentos A, B, C e D, dados na figura 5.6, onde o segmento A exclui o segmento D (neste caso, não está definida a relação D **EX** A).

Segmento	Identificador	r	c	da
A	1	0	60	122
B	2	20	20	121
C	3	30	20	120
D	4	90	20	110

Exclusão:

1 **EX** 4

Figura 5.6 - Exemplo 3: Entrada Fornecida ao EPPE

Após o processamento dos dados de entrada apresentados na figura acima, o **EPPE** gerou um escalonamento ótimo praticável para os segmentos, o qual é apresentado na figura 5.7, através de um diagrama de Gantt. Na obtenção do escalonamento, o **EPPE** gerou 4 nós da árvore de busca. O segmento mais retardado é D, o qual termina sua execução exatamente no seu "deadline", ou seja, $l[D] = 0$.

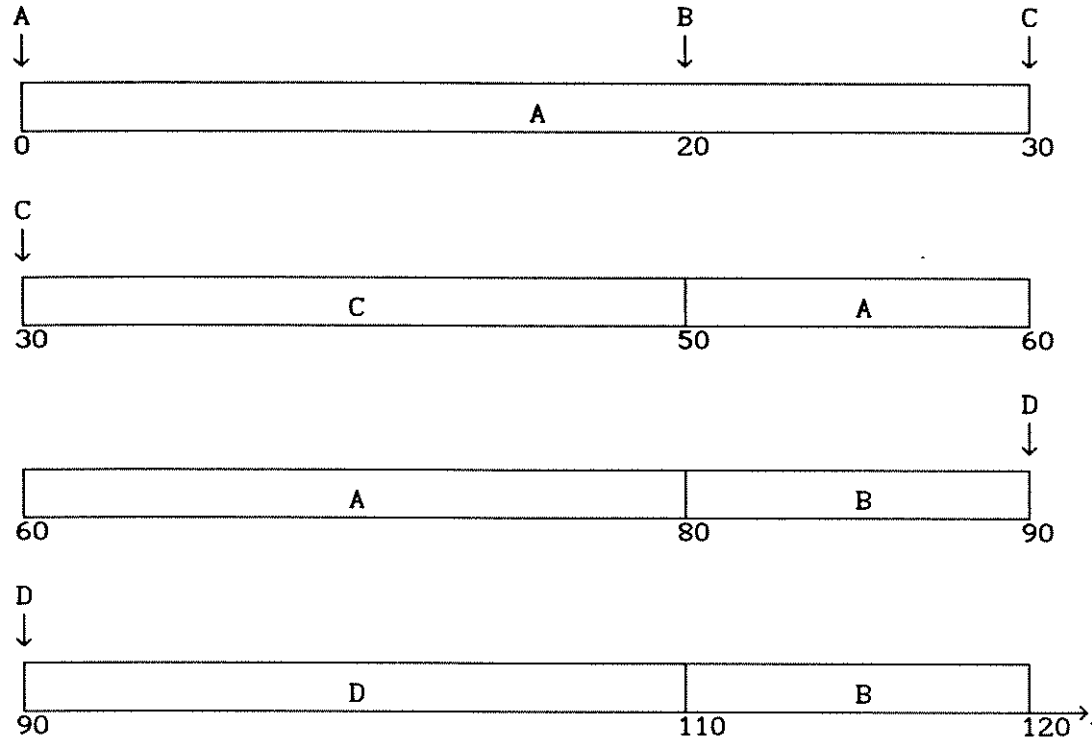


Figura 5.7 - Exemplo 3: Escalonamento "Off-Line" de Segmentos

5.3 - ATENDIMENTO de PROCESSOS ESPORÁDICOS

Nesta seção serão apresentados vários resultados obtidos usando-se os algoritmos propostos para o atendimento "on-line" de processos esporádicos. A estratégia de teste usada para a validação dos algoritmos é a seguinte:

- obtenção de um escalonamento praticável, em modo "off-line", para um determinado conjunto de processos periódicos, usando-se o serviço *Escalonamento off-line* do **EPPE**;
- atendimento "on-line" dos processos esporádicos usando-se o procedimento proposto;
- nesta etapa, considera-se os processos esporádicos como possuidores de tempos de chegadas determinísticos, e faz-se o escalonamento "off-line" do conjunto de processos periódicos inicial acrescido destes novos processos;

- compara-se os resultados obtidos na segunda e terceira etapas.

Em todos os exemplos apresentados a seguir, será usado o escalonamento "off-line" obtido para os processos periódicos do exemplo 1 da seção anterior.

Exemplo 4: considere um processo esporádico *s* chegando no instante 65, com "deadlline" igual a 17 e tempo de execução igual a 8 (*s* = (65, 8, 17)), no cenário da figura 5.3. O "deadline" absoluto do processo esporádico *s* é 65 + 17 = 82. O processo *s* é aceito e o escalonamento resultante é apresentado na figura 5.8.

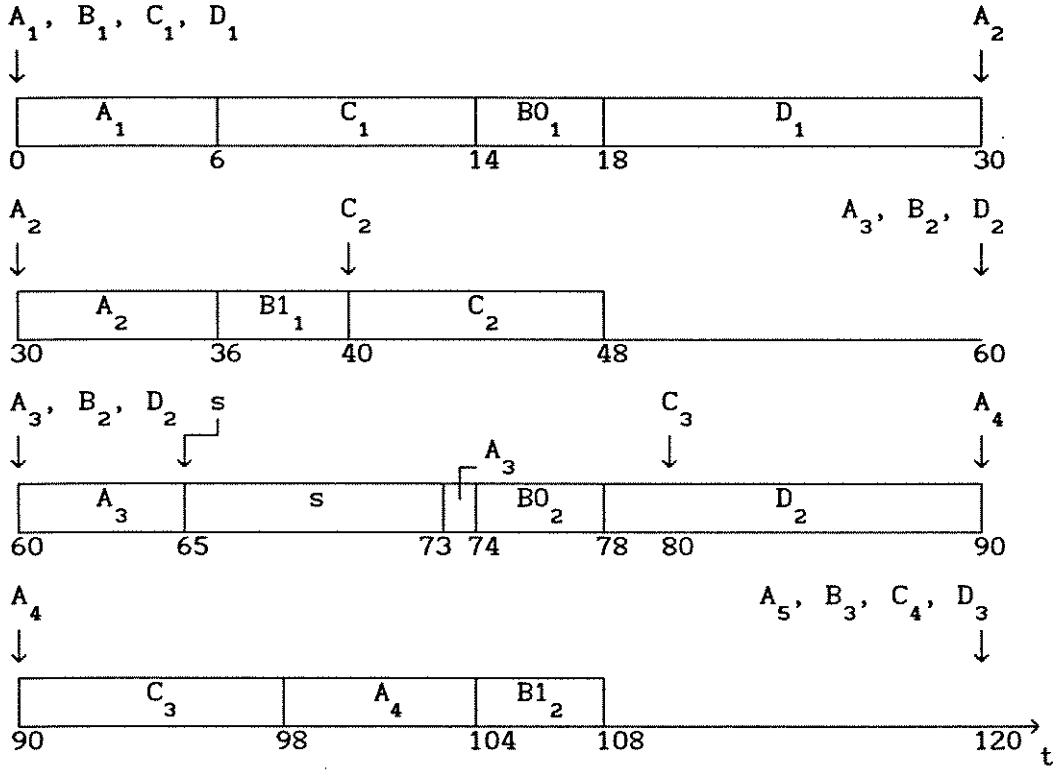


Figura 5.8 - Exemplo 4: Atendimento "On-Line" de Proc. Esporádicos

O processo esporádico *s*, após ser aceito, possui máxima prioridade e ocupa o processador imediatamente sem ser interrompido, pois o processo A_3, que estava executando, pode ser atrasado de 8

unidades de tempo.

Considere o processo esporádico *s* como um 14º segmento do conjunto **S(℘)** apresentado na figura 5.2, cujos parâmetros são: *r*[*s*] = 65, *c*[*s*] = 8, *da*[*s*] = 82, resultando em um novo conjunto **S'(℘)**. Fazendo o escalonamento deste novo conjunto de segmentos usando-se o serviço *Escalonamento off-line* do **EPPE** obteve-se um escalonamento idêntico ao da figura 5.8.

Exemplo 5: considere um processo esporádico *s* chegando no instante 65, com "deadline" igual a 35 e tempo de execução igual a 21 (*s* = (65, 21, 35)), no cenário da figura 5.3. O "deadline" absoluto do processo esporádico *s* é 65 + 35 = 100. Em *t* = 65, quando o processo esporádico chega, é executado o teste de aceitação para este e tem-se que *phi* = *omega*(65, 100) = 20. Logo, como *phi* < 21, o processo esporádico é recusado pelo teste de aceitação, pois caso este fosse aceito comprometeria o escalonamento dos processos periódicos previamente garantidos, e a sequência de escalonamento prossegue idêntica à da figura 5.3.

Considere o processo esporádico *s* como um 14º segmento do conjunto **S(℘)** apresentado na figura 5.2, cujos parâmetros são: *r*[*s*] = 65, *c*[*s*] = 21, *da*[*s*] = 100, resultando em um novo conjunto **S'(℘)**. Fazendo o escalonamento deste novo conjunto de segmentos usando-se o serviço *Escalonamento off-line* do **EPPE**, obteve-se um escalonamento ótimo não praticável para os segmentos de **S'(℘)**, o qual é apresentado na figura 5.9.

O segmento mais retardado no escalonamento da figura 5.9 é $B1_2$, o qual termina 1 unidade de tempo após seu "deadline", ou seja, $l[B1_2] = 1$. Portanto, para este conjunto de segmentos **S'(℘)**, não existe nenhum escalonamento praticável possível. Note que o segmento *s* (antigo processo esporádico *s*) cumpriu o seu "deadline", mas comprometeu o "deadline" de um segmento previamente garantido.

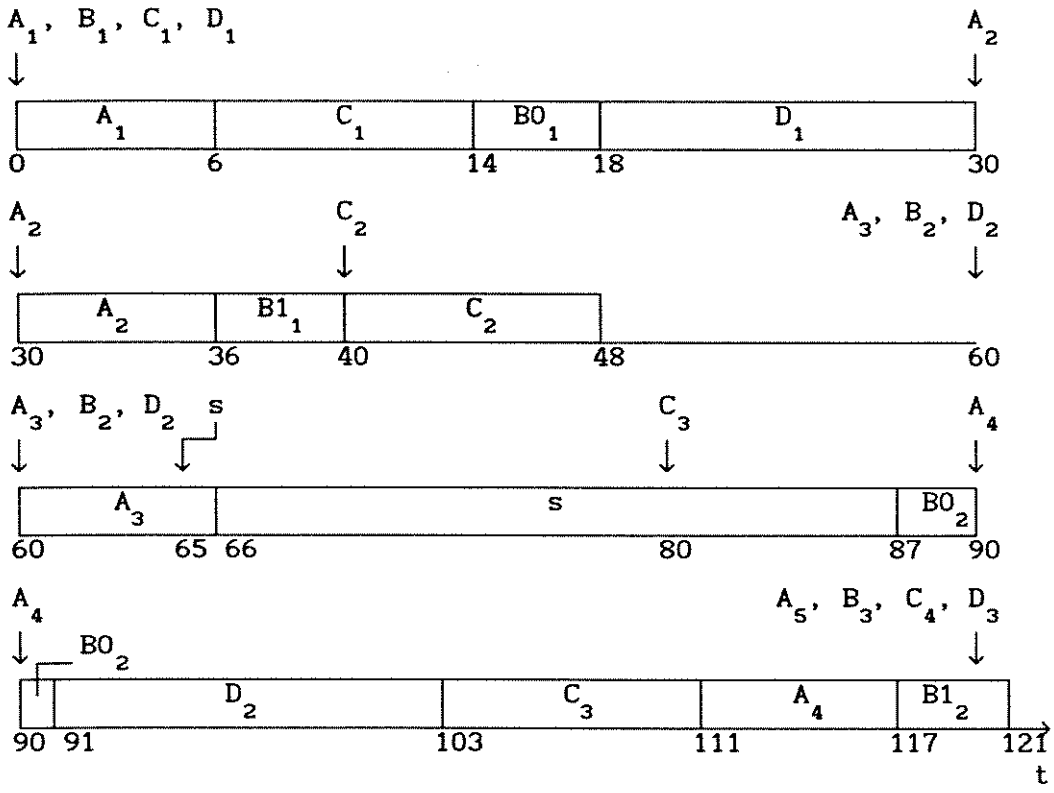


Figura 5.9 - Exemplo 5: Escalonamento "Off-Line" de S'(𝔓)

Exemplo 6: considere um processo esporádico *s* chegando no instante 12, com "deadline" igual a 33 e tempo de execução igual a 10 (*s* = (12, 10, 33)), no cenário da figura 5.3. O "deadline" absoluto do processo esporádico *s* é 12 + 33 = 45. O processo *s* é aceito e o escalonamento resultante é apresentado na figura 5.10.

Considere o processo esporádico *s* como um 14º segmento do conjunto **S(𝔓)** apresentado na figura 5.2, cujos parâmetros são: *r*[*s*] = 12, *c*[*s*] = 10, *da*[*s*] = 45, resultando em **S'(𝔓)**. Fazendo o escalonamento deste novo conjunto de segmentos usando-se o serviço *Escalonamento off-line* do **EPPE**, obteve-se um escalonamento ótimo praticável para os segmentos de **S'(𝔓)**, o qual é apresentado na figura 5.11.

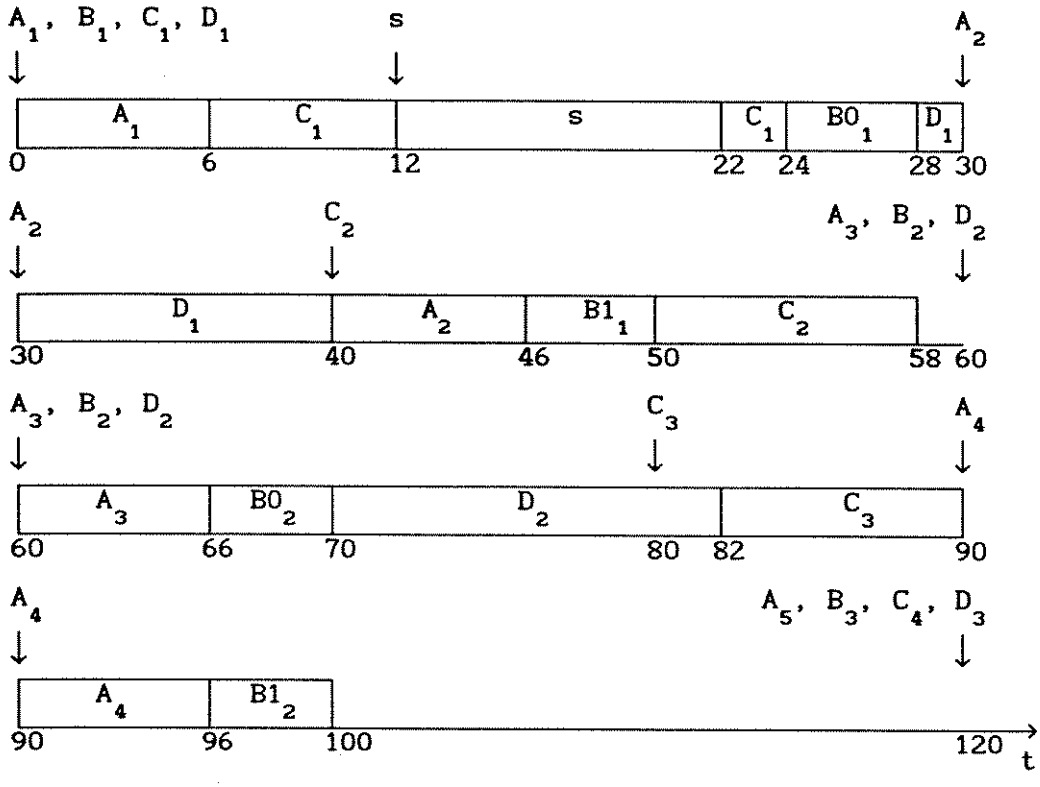


Figura 5.10 - Exemplo 6: Atendimento "On-Line" de Proc. Esporádicos

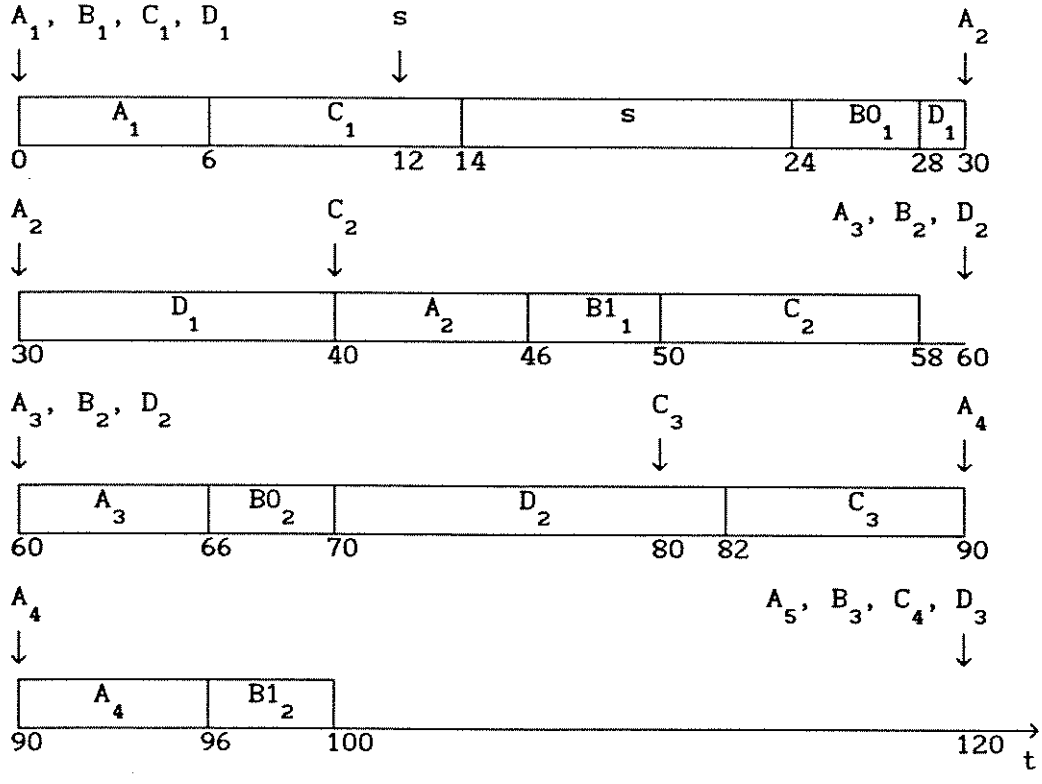


Figura 5.11 - Exemplo 6: Escalonamento "Off-Line" de S'(t)

Nota-se no escalonamento da figura 5.10 que o processo *s* termina sua execução duas unidades de tempo antes, em relação ao escalonamento da figura 5.11. Isto se deve ao fato de que no atendimento "on-line" um processo esporádico recebe prioridade máxima em relação a todos os outros processos, enquanto que no algoritmo de Xu e Parnas[21] a prioridade de um processo é função de seu "deadline" e de relações **PREEMPT**. Observa-se ainda que o algoritmo "off-line" minimiza a quantidade de mudanças de contexto, não preemptando desnecessariamente o processo C_1.

Exemplo 7: considere um processo esporádico *s* chegando no instante 38, com "deadline" igual a 27 e tempo de execução igual a 22 (*s* = (38, 22, 27)), no cenário da figura 5.3. O "deadline" absoluto do processo esporádico *s* é 38 + 27 = 65. O processo *s* é aceito e o escalonamento resultante é apresentado na figura 5.12.

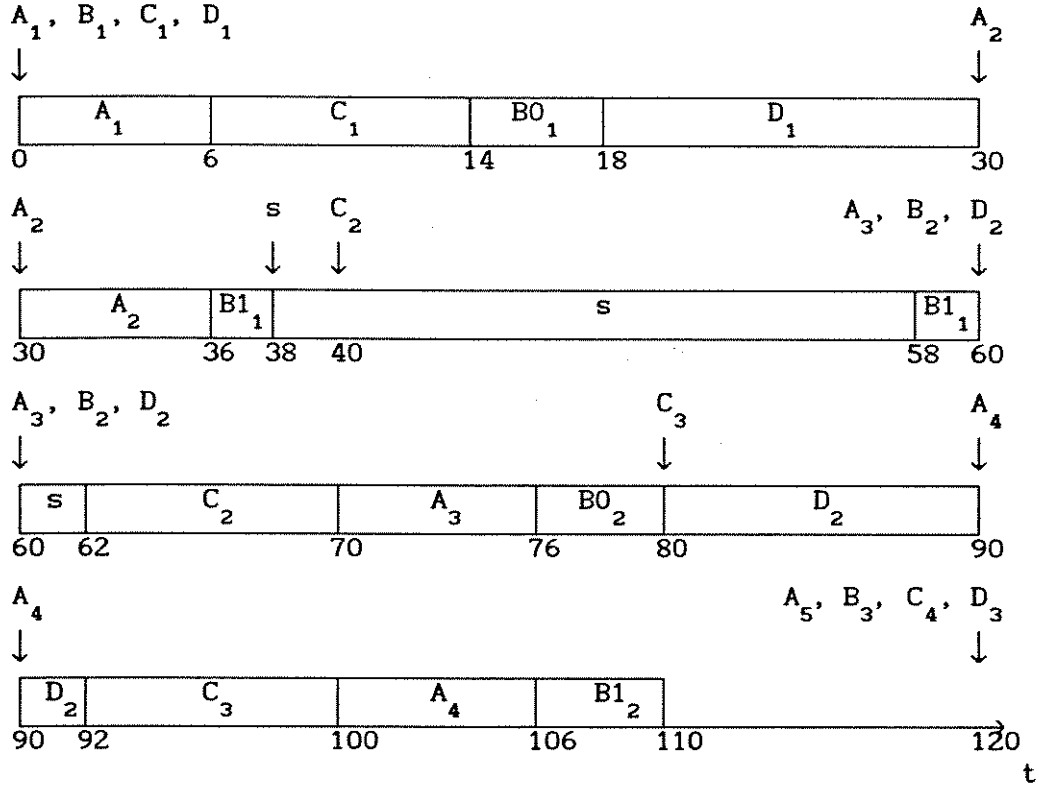


Figura 5.12 - Exemplo 7: Atendimento "On-Line" de Proc. Esporádicos

Considere o processo esporádico *s* como um 14º segmento do conjunto **S(℘)** apresentado na figura 5.2, cujos parâmetros são *r*[*s*] = 38, *c*[*s*] = 22, *da*[*s*] = 65, resultando em **S'(℘)**. Fazendo o escalonamento deste novo conjunto de segmentos usando-se o serviço *Escalonamento off-line* do **EPPE**, obteve-se um escalonamento ótimo praticável para os segmentos de **S'(℘)**, o qual é apresentado na figura 5.13.

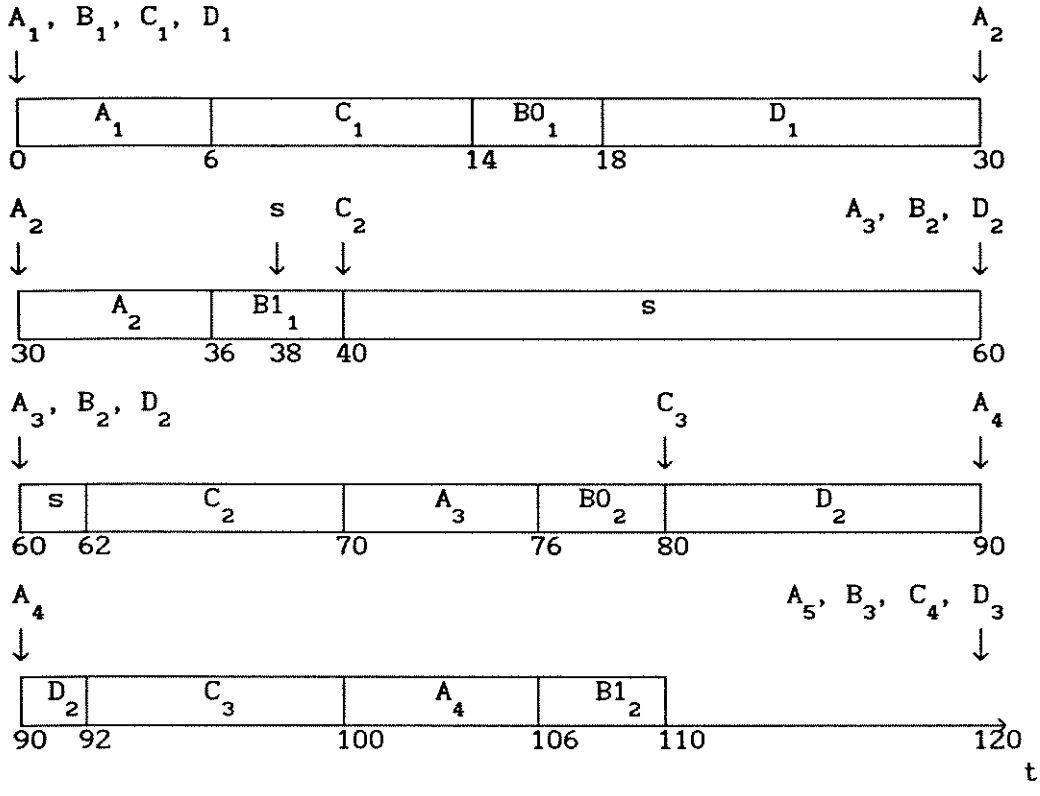


Figura 5.13 - Exemplo 7: Escalonamento "Off-Line" de S'(℘)

Nota-se no escalonamento da figura 5.12 que o processo esporádico *s* ao ser aceito em *t* = 38 ocupa imediatamente o processador, por receber prioridade máxima. Em *t* = 58, quando o atraso permitido do segmento $B1_1$ torna-se 0, o processo *s* é suspenso e $B1_1$ retoma o processador, a tempo de cumprir o seu "deadline".

No escalonamento da figura 5.13, obtido "off-line", apesar do processo *s* iniciar sua execução em *t* = 40, ele termina no mesmo instante (*t* = 62), além de não preemptar $B1_1$.

Exemplo 8: considere dois processos esporádicos *s*1 e *s*2, o primeiro chegando no instante 37, com "deadline" igual a 23 e tempo de execução igual a 4 (*s*1 = (37, 4, 23)) e o segundo chegando no instante 73, com "deadline" igual a 7 e tempo de execução igual a 3 (*s*2 = (73, 3, 7)), ambos no cenário da figura 5.3. Os "deadlines" absolutos dos processos são: *da*[*s*1] = 37 + 23 = 60 e *da*[*s*2] = 73 + 7 = 80. Os processos *s*1 e *s*2 são aceitos e o escalonamento resultante é apresentado na figura 5.14.

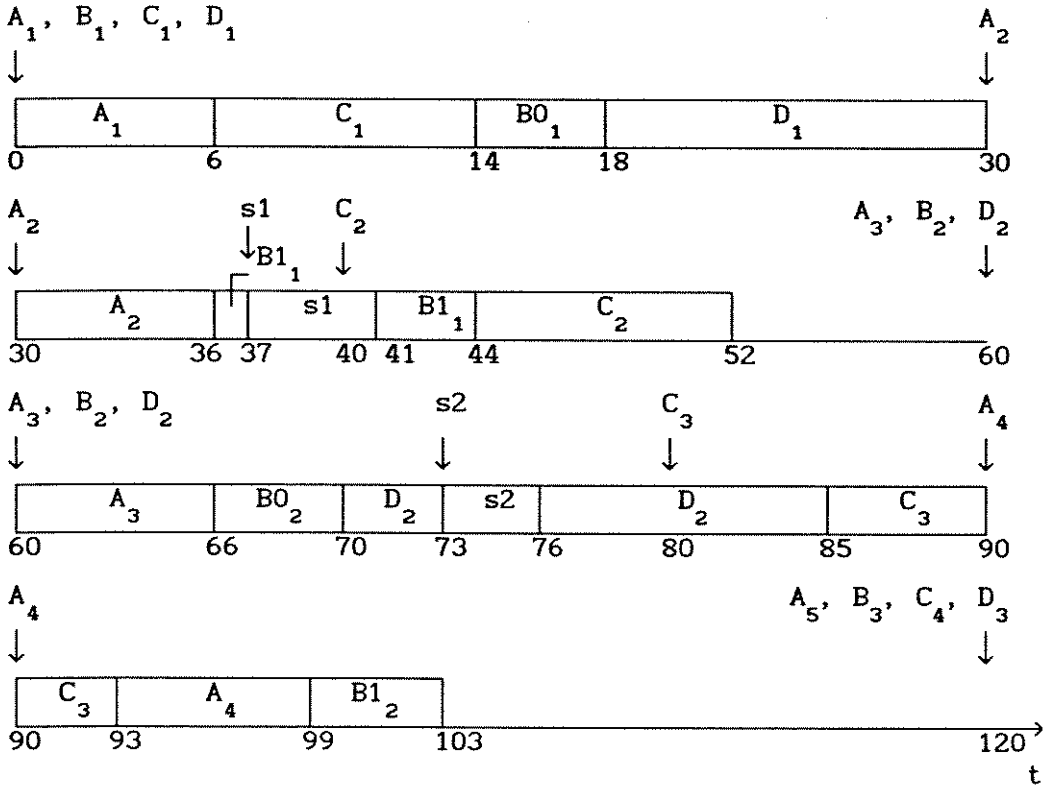


Figura 5.14 - Exemplo 8: Atendimento "On-Line" de Proc. Esporádicos

Considere os processos esporádicos *s*1 e *s*2 como os 14º e 15º segmentos, respectivamente, do conjunto **S(℘)** apresentado na figura 5.2, cujos parâmetros são: *r*[*s*1] = 37, *c*[*s*1] = 4, *da*[*s*1] = 60, e *r*[*s*2] = 73, *c*[*s*2] = 3, *da*[*s*2] = 80, resultando em **S'(℘)**. Fazendo o escalonamento deste novo conjunto de segmentos usando-se o serviço *Escalonamento off-line* do **EPPE**, obteve-se um escalonamento ótimo praticável para os segmentos de **S'(℘)**, o qual é apresentado na figura

5.15.

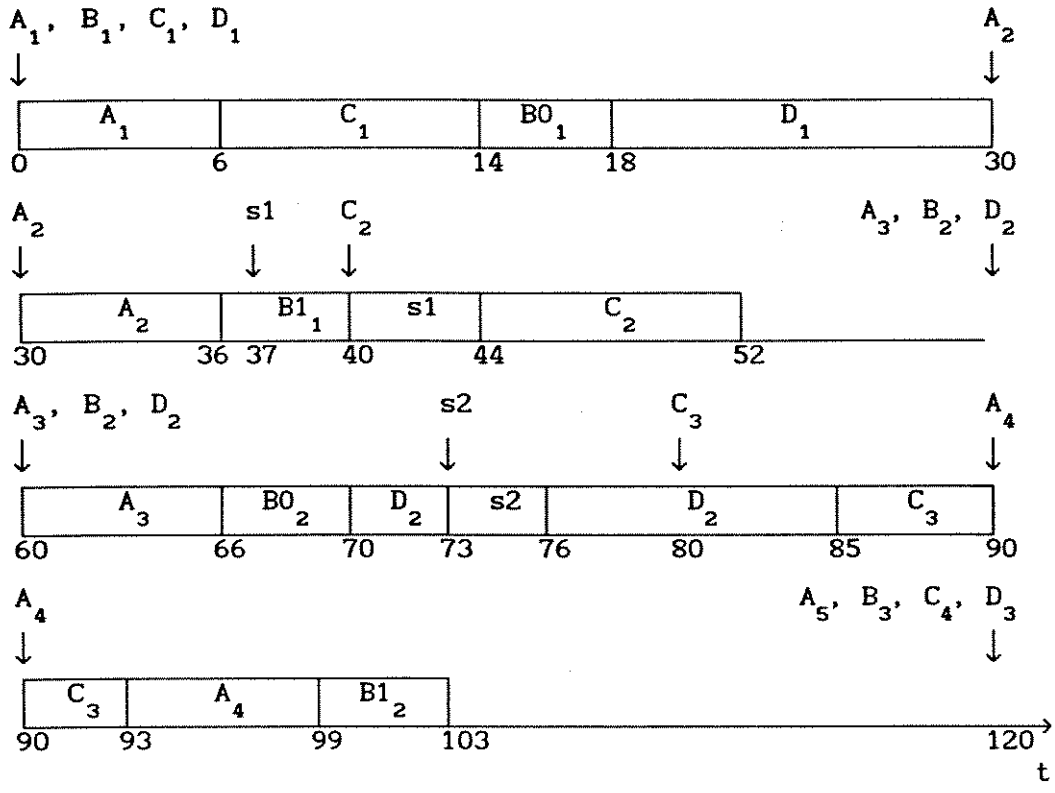


Figura 5.15 - Exemplo 8: Escalonamento "Off-Line" de S'(ℙ)

Nota-se no escalonamento da figura 5.14 que o processo esporádico $s1$ termina sua execução três unidades de tempo antes, em relação ao escalonamento da figura 5.15, à custa de uma preempção no segmento $B1_1$. No escalonamento "off-line" de **S'(ℙ)**, o processo $s2$ também preempta o segmento D_2, devido ao fato do "deadline" de D_2 ($da[D_2] = 120$) ser maior que o "deadline" de $s2$ ($da[s2] = 80$).

Exemplo 9: considere dois processos esporádicos $s1$ e $s2$, o primeiro chegando no instante 5, com "deadline" igual a 18 e tempo de execução igual a 7 ($s1 = (5, 7, 18)$) e o segundo chegando no instante 7, com "deadline" igual a 13 e tempo de execução igual a 10 ($s2 = (7, 10, 13)$), ambos no cenário da figura 5.3. Os "deadlines" absolutos dos processos são: $da[s1] = 5 + 18 = 23$ e $da[s2] = 7 + 13 = 20$. O processo $s1$ ao chegar em $t = 5$ é aceito, preempta o segmento A_1 e

começa a executar. O processo *s*2 chega em $t = 7$ e neste instante o processo *s*1 ainda não terminou sua execução. O teste de aceitação para o processo *s*2 é executado e tem-se que *phi*(7, 20) = 13. Como a soma do tempo de execução restante do processo *s*1, cujo valor é 5, com o tempo de execução de *s*2 resulta em 15 (maior que *phi*(7, 20)), o processo *s*2 é recusado pelo teste de aceitação, e o processo *s*1 prossegue sua execução.

Considere os processos esporádicos *s*1 e *s*2 como os 14º e 15º segmentos, respectivamente, do conjunto **S(𝔓)** apresentado na figura 5.2, cujos parâmetros são: *r*[*s*1] = 5, *c*[*s*1] = 7, *da*[*s*1] = 23, e *r*[*s*2] = 7, *c*[*s*2] = 10, *da*[*s*2] = 20, resultando em **S'(𝔓)**. Fazendo o escalonamento deste novo conjunto de segmentos usando-se o serviço *Escalonamento off-line* do **EPPE**, obteve-se um escalonamento ótimo praticável para os segmentos de **S'(𝔓)**, o qual é apresentado na figura 5.16.

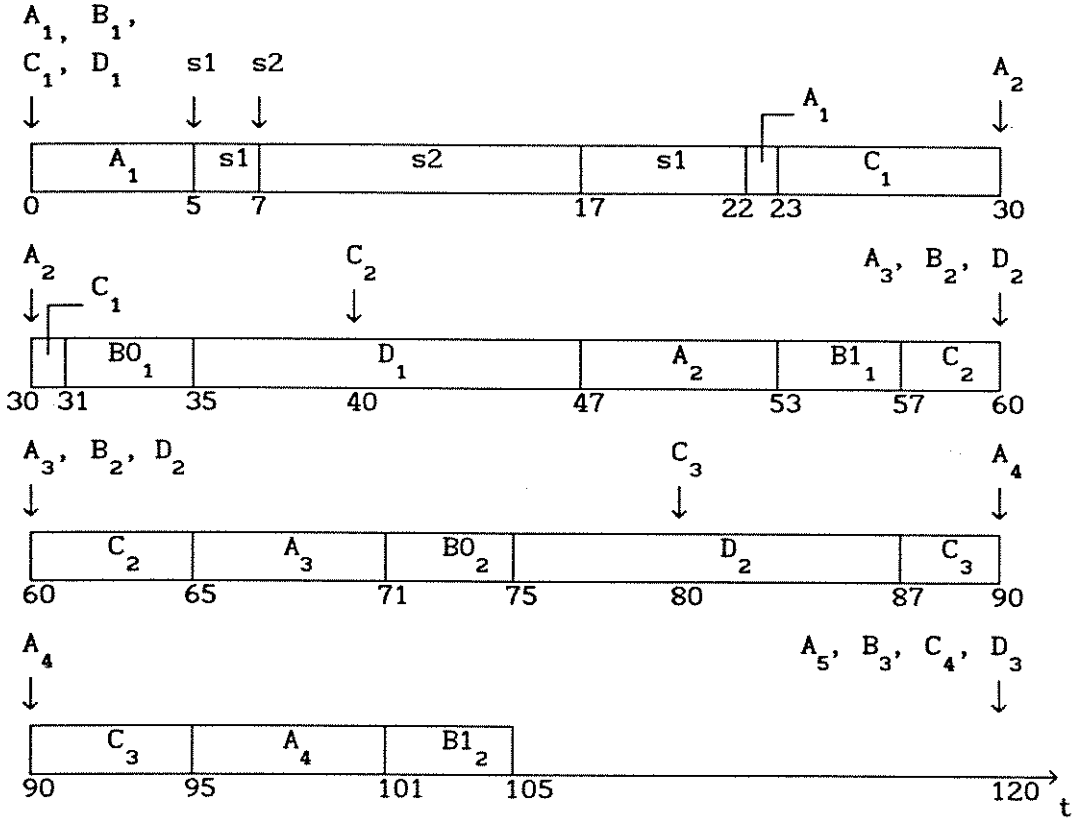


Figura 5.16 - Exemplo 9: Escalonamento "Off-Line" de S'(𝔓)

No exemplo 9, a política de atendimento "on-line" de processos esporádicos não consegue escalonar o processo esporádico $s2$, enquanto que o algoritmo "off-line" de Xu e Parnas[21], com o conhecimento prévio dos tempos de chegadas dos processos $s1$ e $s2$, consegue. Logo, a política de atendimento "on-line" de processos esporádicos não é ótima. Uma forma de reduzir a limitação do algoritmo "on-line" proposto neste trabalho é a utilização de múltiplas versões para processos esporádicos.

5.4 - CONSIDERAÇÕES FINAIS

A política de atendimento "on-line" não permite preempções entre processos esporádicos, ou seja, um processo esporádico que chega no sistema só poderá iniciar execução, caso venha a ser aceito, após o término de todos os processos esporádicos previamente aceitos. Se por um lado tal prática fornece uma garantia adicional aos processos esporádicos já garantidos e diminui os custos de mudança de contexto, ela pode impedir que novos processos esporádicos que cheguem, com "deadlines" mais urgentes, sejam aceitos. No exemplo 9, o processo $s2$ só pode ser aceito caso seja possível a preempção do processo $s1$ pelo processo $s2$.

A estratégia de escalonamento ótima para os processos esporádicos é o "earliest deadline". Contudo, esta estratégia não foi utilizada devido ao fato do teste de aceitação, no caso de existirem vários processos esporádicos aceitos aguardando execução, se tornar bastante complexo para ser executado "on-line", pois todos os processos esporádicos previamente aceitos que não terminaram suas execuções teriam que ser novamente testados quando da chegada de um novo processo esporádico com "deadline" mais urgente.

Em situações nas quais não existem processos esporádicos previamente garantidos quando um processo esporádico chega, a política de atendimento "on-line" é ótima, podendo até fornecer um tempo de resposta para o processo esporádico melhor que o algoritmo "off-line" de Xu e Parnas[21], uma vez que o processo esporádico irá executar com alta prioridade, atrasando o máximo

possível os blocos de processos periódicos.

CAPÍTULO 6

CONCLUSÕES

CONCLUSÕES

Propõe-se neste trabalho uma estratégia para o escalonamento de processos periódicos e esporádicos em STRC monoprocessados, os quais devem satisfazer diversas restrições. Na estratégia proposta, os processos periódicos podem estar sujeitos à restrições de tempo de pronto, "deadline", relações de precedência e relações de exclusão mútua. O problema de escalonar um conjunto de processos em um sistema monoprocessador sujeito à estas restrições é conhecido ser "NP-hard"[21], [22], o que efetivamente impede o escalonamento destes em modo "on-line".

Os processos periódicos, que realizam a maior parte da computação em STRC[24], possuem tempos de pronto das diversas instâncias de cada processo conhecidos de antemão, o que possibilita o escalonamento em modo "off-line". Para o escalonamento destes, usou-se um algoritmo "off-line" ótimo, disponível na literatura[21].

Para os processos esporádicos, que possuem tempos de pronto não determinísticos, uma vez que estes estão geralmente associados à eventos aleatórios do sistema cujos instantes de ocorrência são imprevisíveis, propõe-se um procedimento inédito para o atendimento "on-line" destes. O procedimento faz uso de informações extraídas, também em modo "off-line", do escalonamento de processos periódicos obtido previamente. A estratégia "on-line" considera restrições de tempo de pronto e "deadline". Processos esporádicos não podem possuir relações de precedência com outros processos, pois isto implicaria em uma completa ausência de determinismo do sistema.

Existe ainda muito trabalho a ser feito na área de escalonamento de processos em STRC, mesmo em ambientes monoprocessados. A seguir, enumera-se algumas sugestões para o prosseguimento deste trabalho:

- estender a política de atendimento "on-line" para considerar processos esporádicos com restrições de exclusão mútua;

- estender o algoritmo "off-line" de Xu e Parnas[21] e a política de atendimento "on-line" proposta para sistemas multiprocessadores e distribuídos;
- estender a política de atendimento "on-line" para permitir preempção entre processos esporádicos já garantidos e processos que chegam no sistema com "deadlines" mais urgentes;
- estender o algoritmo de Xu e Parnas[21] e a política de atendimento "on-line" proposta para considerar requisições de recursos adicionais;
- incluir mecanismos de decisão na política de atendimento "on-line" para que, no caso de um processo ser recusado pelo teste de aceitação, escolher-se a melhor ação a ser tomada em função do tempo disponível, como por exemplo, executar-se tratadores de exceção e/ou outras versões funcionais do processo recusado;
- estudar implementações eficientes para as políticas "on-line" e "off-line".

Algumas das sugestões citadas acima para aumentar a funcionalidade da política de atendimento "on-line" podem ser alcançadas extraindo-se mais informações, em modo "off-line", do escalonamento obtido para os processos periódicos, e organizando-as em estruturas de dados adequadas. Por exemplo, para estender-se a política de atendimento "on-line" a fim de suportar restrições de exclusão mútua entre processos periódicos e esporádicos, poderia-se tentar extrair do escalonamento obtido para os processos periódicos informações de regiões onde cada processo esporádico pode ser escalonado sem incorrer na violação das restrições de exclusão mútua.

Processos aperiódicos que não possuem restrições de "deadline" também podem ser suportados pela estratégia proposta. Basta fornecer-se a estes processamento em modo "background", ou seja, sempre que não existir processos periódicos ou esporádicos prontos para executar.

Devido à complexidade dos problemas de escalonamento em **STRC**, tipicamente "NP-hard"[22], [23], e da escassez de tempo disponível para um escalonador "on-line" computar escalonamentos dinamicamente, o uso de escalonamento "off-line" torna-se essencial

para garantir as rígidas restrições temporais encontradas nestes sistemas. Na estratégia proposta, procurou-se executar o maior número possível de procedimentos em modo "off-line" (escalonamento de processos periódicos, cálculo dos atrasos máximos e da função *omega*), para que a parte da estratégia de escalonamento global executada "on-line" seja simples e eficiente. Por isso, acredita-se que a estratégia proposta pode ter uma larga aplicação em situações reais, principalmente nos níveis de **STRC** mais próximos dos processos controlados, onde as restrições de tempo são bastante rígidas.

A política de atendimento "on-line" de processos esporádicos proposta é independente da política de escalonamento "off-line", podendo vir a ser usada com outros algoritmos de escalonamento "off-line". Optou-se por utilizar o algoritmo dado em [21] por este ser ótimo e o mais abrangente disponível na literatura no momento.

Finalmente, cumpre ressaltar, que uma estratégia de atendimento "on-line" ótima de processos esporádicos sujeitos à restrições de tempo de pronto, "deadline" e relações de exclusão mútua em **STRC** é impraticável, a menos que P = NP[22].

BIBLIOGRAFIA

[01] - Liu, C.L. & Layland, J.W., (1973). "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", Journal of the Association for Computing Machinery, Vol. 20, nº 1:46-61.

[02] - Mok, A.K., (1983). "Fundamental Design Problems of Distributed Systems for the Hard-Real-Time Environment", Ph.D Thesis, Department of Electrical Engineering and Computer Science, M.I.T., Cambridge.

[03] - Ben-Ari, M., (1990). "Principles of Concurrent and Distributed Programming", UK: Prentice Hall International Ltd..

[04] - Cheng, S.C. & Stankovic, J.A., (1987). "Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey", Real-time Systems Newsletter, Vol. 3, nº 2:1-24.

[05] - Sprunt, B., Sha, L. & Lehoczky, J., (1989). "Aperiodic Task Scheduling for Hard-Real-Time Systems", The Journal of Real-Time Systems, Vol. 1, nº 1:27-60.

[06] - Sha, L. & Goodenough, J.B., (1990). "Real-Time Scheduling - Theory and Ada", Computer, April:53-62.

[07] - Lehoczky, J., Sha, L. & Ding, Y., (1989). "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior", Proceedings of IEEE Real-Time Systems Symposium, Los Alamitos:166-171.

[08] - Sha, L., Lehoczky, J.P. & Rajkumar, R., (1986). "Solutions for Some Practical Problems in Prioritzed Preemptive Scheduling", Proceedings of IEEE Real-Time Systems Symposium, New Orleans:181-191.

[09] - Lehoczky, J.P., Sha, L. & Strosnider, J.K., (1987). "Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", Proceedings of IEEE Real-Time Systems Symposium, San Jose:261-270.

[10] - Sha, L., Rajkumar, R. & Lehoczky, J.P., (1987). "Priority Inheritance Protocols: An Approach to Real-Time Synchronization", technical report, Department of Computer Science, Carnegie Mellon University.

[11] - Rajkumar, R., Sha, L., Lehoczky, J.P. & Ramamritham, K., (1988). "An Optimal Priority Inheritance Protocol for Real-Time Synchronization", technical report, Carnegie Mellon University and University of Massachusetts.

[12] - Rajkumar, R., Sha, L. & Lehoczky, J.P., (1988). "Real-Time Synchronization Protocols for Multiprocessors", Proceedings of IEEE Real-Time Systems Symposium, Los Angeles:259-269.

[13] - Liu, J.W.S., Lin, K.J. & Natarajan, S., (1987). "Scheduling Real-Time, Periodic Jobs Using Imprecise Results", Proceedings of the IEEE Real-Time Systems Symposium, San Jose:252-260.

[14] - Chung, J.Y., Liu, J.W.S. & Lin, K.J., (1990). "Scheduling Periodic Jobs that Allow Imprecise Results", IEEE Transactions on Computers, Vol. 39, nº 9:1156-1174.

[15] - Sha, L., Rajkumar, R., Lehoczky, J. & Ramamritham, K., (1989). "Mode Change Protocols for Priority-Driven Preemptive Scheduling", technical report, Carnegie Mellon University and University of Massachusetts.

[16] - Sha, L., Klein, M.H. & Goodenough, J.B., (1991). "Rate Monotonic Analysis for Real-Time Systems", technical report, Software Engineering Institute, Carnegie Mellon University.

[17] - Mok, A.K., (1984). "The Design of Real-Time Programming Systems Based on Process Models", Proceedings of IEEE Real-Time Systems Symposium: 5-17.

[18] - Chen, M.I. & Lin, K.J., (1990). "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems", The Journal of Real-Time Systems, Vol. 2, nº 4:325-346.

[19] - Hsieh, Y.Z., (1991). Estudo e Análise de Algoritmos de Escalonamento para Aplicações de Tempo Real Crítico, Dissertação de Mestrado, DCA/FEE/UNICAMP.

[20] - Spanó, R.M., (1991). SSE: Proposta de Uma Ferramenta de Software Dedicada ao Estudo de Algoritmos de Escalonamento para Sistemas de Tempo Real Crítico, Dissertação de Mestrado, DCA/FEE/UNICAMP.

[21] - Xu, J. & Parnas, D.L., (1990). "Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations", IEEE Transactions on Software Engineering, Vol. 16, nº 3:360-369.

[22] - Garey, M.R. & Johnson, D.S., (1979). "Computers and Intractability: A Guide to the Theory of NP-Completeness", San Francisco, CA: Freeman.

[23] - Xu, J. & Parnas, D.L., (1991). "On Satisfying Timing Constraints in Hard-Real-Time Systems", Software Engineering Notes, Vol. 16, nº 5:132-146.

[24] - Faulk, S.R. & Parnas, D.L., (1988). "On Synchronization in Hard-Real-Time Systems", Communications of the ACM, Vol. 31, nº 3:274-287.

[25] - French, S., (1986). "Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop", New York: Ellis Horwood Limited.

[26] - Shepard, T. & Gagné, J.A.M., (1991). "A Pre-Run-Time Scheduling Algorithm for Hard Real-Time Systems, IEEE Transactions on Software Engineering, Vol. 17, nº 7:669-677.

[27] - Melo Jr., A., Magalhães, M.F. & Coello, J.M.A., (1992). Escalonamento *On-Line* de Processos Esporádicos em Sistemas de Tempo Real Crítico, Anais do 9º Congresso Brasileiro de Automática, Vol 1:472-476.

[28] - Leung, J.Y.T. & Merril, M.L., (1980). "A Note on Preemptive Scheduling of Periodic, Real-Time Tasks", Information Processing Letters, Vol. 11, nº 3:115-118.

[29] - Chetto, H. & Chetto, M., (1989). "Some Results of the Earliest Deadline Scheduling Algorithm", IEEE Transactions on Software Engineering, Vol. 15, nº 10:1261-1269.

[30] - Ramamritham, K., Stankovic, J.A. & Zhao, W., (1988). "Distributed Scheduling of Tasks with Deadlines and Resource Requirements", technical report, Department of Computer and Information Science, University of Massachusetts.

[31] - Stankovic, J.A., Ramamritham, K. & Cheng, S., (1985). "Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems", IEEE Transactions on Computers, Vol. C-34, nº 12:1130-1143.

[32] - Zhao, W., Ramamritham, K. & Stankovic, J.A., (1987). "Scheduling Tasks with Resource Requirements in Hard Real-Time Systems", IEEE Transactions on Software Engineering, Vol. SE-13, nº 5:564-577.

APÊNDICE A

APÊNDICE A

O seguinte procedimento calcula uma solução válida na qual as restrições de tempo de pronto e os conjuntos de relações de exclusão, precedência e preempção são satisfeitas[21]:

```
lastt := -1 /* Pode ser qualquer valor negativo */
lastseg := 1 /* Pode ser qualquer segmento */
idle := true
Para i := 1 até nseg faça
início
  started[i] := false
  completed[i] := false
  comptimeleft[i] := c[i]
  s[i] := -1
fim
t := 0
Enquanto ¬(∀ i: completed[i] = true) faça
início
  t := min{t | t > lastt ∧ ((∃ i: t = r'[i]) ∨
           ((idle = false) ∧
            (comptimeleft[lastseg] = t - lastt)))}
  Se (idle = false)
  então
  início
    Na solução válida computada pelo procedimento,
    deixe o segmento lastseg executar de lastt até t
    comptimeleft[lastseg] := comptimeleft[lastseg] -
                             (t - lastt)
```

```
        Se (comptimeleft[lastseg] = 0)
        então
        início
          completed[lastseg] := true
          e[lastseg] := t
        fim
      fim
      S := {j | j é elegível em t ∧
            ¬(∃ i: i é elegível em t ∧ (i PM j))}
      Se (S = ø)
      então
        idle := true
      senão
      início
        idle := false
        S1 := {j | da[j] = min{da[i] | i ∈ S}}
        x := segmento que satisfaz c[x] = max{c[i] |
                                              i ∈ S1}
        Se ¬(started[x])
        então
        início
          started[x] := true
          s[x] := t
        fim
        lastseg := x
      fim
      lastt := t
    fim
```

No algoritmo, temos as seguintes variáveis:

- *t*: instante atual no qual se tenta selecionar um segmento para execução;
- *lastt*: último instante no qual se tentou selecionar um segmento para execução;
- *lastseg*: último segmento selecionado para

execução;

- *idle*: indica se houve algum segmento selecionado em *lastt*;
- *started*[*i*]: indica se o segmento *i* já iniciou execução;
- *completed*[*i*]: indica se o segmento *i* já completou sua execução;
- *comptimeleft*[*i*]: tempo de execução restante do segmento *i*;
- *s*[*i*]: tempo de início de execução do segmento *i*;
- *r'*[*i*]: tempo de pronto ajustado do segmento *i*;
- *e*[*i*]: tempo de término de execução do segmento *i*;
- *nseg*: número de segmentos a ser escalonado.

APÊNDICE B

APÊNDICE B

O seguinte procedimento encontra uma solução válida praticável ou ótima usando a técnica "branch and bound"[21]:

```
nodeindex := 0
inicializar(PC(nodeindex), EX)
PM(nodeindex) := ø
opennodeset := ø
optimal := false
feasible := false
Se (consistent(PC(nodeindex), EX, PM(nodeindex)))
então
início
  schedule(nodeindex) :=
    validsolution(PC(nodeindex), EX, PM(nodeindex))
  leastlowerbound:= lowerbound(nodeindex)
  Se (lateness(nodeindex) = leastlowerbound)
  então
    optimal := true
  Se (lateness(nodeindex) ≤ 0)
  então
    feasible := true
  Se ¬(optimal)
  então
  início
    opennodeset := {nodeindex}
    minlateness := lateness(nodeindex)
    minlatenode := nodeindex
```

```
Enquanto ¬(optimal) faça
início
  lowestboundset := {l ∈ opennodeset |
                lowerbound(l) = leastlowerbound}
  parentnode := nó de lowestboundset que
    satisfaz: lateness(parentnode) =
    mim{lateness(i) | i ∈ lowestboundset}
  j := latestsegment(schedule(parentnode))
  firstchildnode := nodeindex + 1
  Calcular conjunto G1(parentnode)
  Para cada segmento k ∈ G1(parentnode) faça
  início
    nodeindex := nodeindex + 1
    Nó filho nodeindex herda todas as relações
        PC, EX, PM do nó parentnode
    PC(nodeindex) := PC(nodeindex) ∪ {(j PC k)}
  fim
  Calcular conjunto G2(parentnode)
  Para cada segmento k ∈ G2(parentnode) faça
  início
    nodeindex := nodeindex + 1
    Nó filho nodeindex herda todas as relações
        PC, EX, PM do nó parentnode
    Para todo segmento l tal que (k EX l)
          e l executa entre k e j em
          schedule(parentnode) faça
      PC(nodeindex) := PC(nodeindex) ∪
                          {(l PC k)}
    Para todo segmento q tal que ¬(k EX q)
          e q executa entre k e j em
          schedule(parentnode) faça
      PM(nodeindex) := PM(nodeindex) ∪
                          {(q PM k)}
    PM(nodeindex) := PM(nodeindex) ∪ {(j PM k)}
  fim
  opennodeset := opennodeset - {parentnode}
```

```
Para childnode := firstchildnode até
     nodeindex faça
início
  Se consistent(PC(childnode), EX,
                PM(childnode))
  então
  início
    schedule(childnode) :=
      validsolution(PC(childnode), EX,
                    PM(childnode))
    Se (lateness(childnode) < minlateness)
    então
    início
      minlateness := lateness(childnode)
      minlatenode := childnode
    fim
    Se (lateness(childnode) ≤ 0)
    então
      feasible := true
    Se (lowerbound(childnode) < minlateness)
    então
      opennodeset := opennodeset ∪ {childnode}
  fim
fim
Se (opennodeset = ø)
então
  optimal := true
senão
início
  leastlowerbound := min{lowerbound(i) |
                         i ∈ opennodeset}
  Se (leastlowerbound ≥ minlateness)
  então
    optimal := true
fim
fim
```

```
            minlateschedule := schedule(minlatenbode)
        fim
    fim
```

No algoritmo, um nó em *opennodeset* é um nó que não possui nós filhos, mas que pode ser selecionado para se expandir a árvore de busca a partir deste, ou seja, eventualmente pode vir a ser um nó pai. **PC**(*nodeindex*) e **PM**(*nodeindex*) são, respectivamente, os conjuntos de relações de precedência e preempção associados com o nó identificado por *nodeindex*. **EX** é o conjunto constante de relações de exclusão. *schedule*(*nodeindex*) é a solução válida computada usando o algoritmo apresentado no apêndice A, fornecendo **PC**(*nodeindex*), **EX** e **PM**(*nodeindex*). *lateness*(*nodeindex*) é o retardo do escalonamento de *schedule*(*nodeindex*). *lowerbound*(*nodeindex*), **G1**(*nodeindex*) e **G2**(*nodeindex*) são, respectivamente, o "lowerbound" e os conjuntos **G1** e **G2**, computados a partir de *schedule*(*nodeindex*). O procedimento *inicializar*(**PC**(*nodeindex*), **EX**) inicializa o conjunto **PC** e **EX**, respectivamente, com as relações de precedência e exclusão mútua que devem ser satisfeitas no problema original. O conjunto **PC** deve receber, adicionalmente, relações de precedência que garantam a ordenação dos segmentos dentro de cada processo.

Para obter-se maior eficiência no processo de busca, ao invés do algoritmo terminar somente quando um escalonamento ótimo for encontrado, o algoritmo pode ser facilmente alterado para terminar assim que um escalonamento praticável for encontrado. Deve-se observar, contudo, que escalonamentos ótimos não são necessariamente praticáveis.