UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE ENGENHARIA DA COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Estudo e Análise de Algoritmos de Escalonamento para Aplicações de Tempo Real Crítico

Este exemplar corresponde à redação final da tese defendida por Yi Zung Hsieh e aprovada pela Comissão Julgadora em 29/09/1991
Orientador

BC/91024473

POR : YI ZUNG HSIEH

ORIENTADOR : MAURÍCIO F. MAGALHÃES

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

- Março de 1991 -

UNICAMP
BIBLIOTECA CENTRAL

Dedico este trabalho aos meus pais

宗成 e 玉金

aos meus irmãos

孟時 e 政霖

e ao Yu

淡泊以明志
寧靜以致遠

AGRADECIMENTOS

- Ao professor Maurício F. Magalhães, pela orientação e constante incentivo ao longo do desenvolvimento deste trabalho.

- Ao CNPQ, pelo auxílio financeiro

- Aos meus companheiros de estrada Marcélia, Roberto Higa e Sergio Braga, cuja amizade e convívio me proporcionaram momentos inesquecíveis nessa fase da minha vida.

- Ao Yu, pela dedicação, carinho e confiança; em especial pela paciente correção sintática e semântica da dissertação.

- Ao Hélio, pela suporte da ferramenta usada durante testes.

- Aos colegas Rodrigo Spanó, Paulo Minoru, Paulo Maurício, Adson, Sakahi, Humberto e Ruy, pela grande amizade e companheirismo.

- Ao Wu, Alencar, Juan e Sibelius, pelas valiosas discussões e críticas construtivas sobre este trabalho.

- Ao Victor Sanchez, meu companheiro de sala, e às minhas amigas Chen e Chang, pela compreensão e apóio moral.

- A minha família, por tudo que fizeram e ainda fazem por mim.

- A todos aqueles que direta ou indiretamente contribuiram para a realização deste trabalho.

RESUMO

As restrições de temporização dos processos num sistema de tempo real crítico (Hard Real Time System - HRTS) é um fator da máxima importância; os problemas de escalonamento geralmente envolvem a garantia da exequibilidade dentro das limitações impostas pelas aplicações e, conforme a relação interprocessos dentro de um sistema, aumenta-se o grau de complexidade do escalonador.

Neste trabalho, será realizada uma série de análises sobre os problemas verificados nos escalonadores de HRTS exercitados em seis ambientes diferentes.

Os estudos estão concentrados sobre dois escalonadores representantes de duas categorias diferentes : o escalonador estático "Taxa Monotônica" e o escalonador dinâmico "Earliest Deadline"; seus desempenhos são discutidos e questionados de diversas maneiras durante as respectivas apresentações; algumas propostas também serão levantadas visando a melhoria dos algoritmos de escalonamento.

ÍNDICE

1. Introdução 1

2. Bases Teóricas e Conceitos Fundamentais 3
2.1. Definição de Hard Real Time System - HRTS 3
2.2. Processos em HRTS 3
2.2.1. Definição 3
2.2.2. Tipos de Processos 5
2.2.3. Relação entre Processos 6
2.3. Escalonadores em HRTS 7
2.3.1. Definição e Características 7
2.3.2. Ambientes de Monoprocessamento, Multipprocessamento e Sistemas Distribuídos 8
2.3.3. O algoritmo de Escalonamento "Prioridade Dirigida" e "Preemptivo" 9
2.3.3.1. Escalonadores Estáticos 10
2.3.3.2. Escalonadores Dinâmicos 10
2.4. Diagrama para as Análises do Trabalho 11
2.5. Ferramentas para Testes de Algoritmos 13
2.5.1. Núcleo de Tempo Real DEADMOSI 14
2.5.1.1. Características Gerais do DEADMOSI 15
2.5.2. Simulador SSE (Sistema de Simulação do Escalonador) 16
2.5.2.1. Características Gerais do SSE 17

3. Escalonador Estático "Taxa Monotônica" 18
3.1. Processos Periódicos e Independentes (AMB 1) 18
3.2. Processos Periódicos e Dependentes (AMB 2) 20
3.2.1. Problema da Inversão de Prioridade 20
3.2.1.1. Definição do Protocolo de Herança de Prioridade 23
3.2.1.2. Algoritmo do Protocolo 23
3.2.1.3. Testes e Implementação 27
3.2.1.3.1. Proposta da Implementação 28
3.2.1.4. Lemas e Teoremas 35
3.2.2. Problemas de "Deadlock" e Cadeia de Bloqueios 37
3.2.2.1. Definição do Protocolo de Prioridade Topo 41

3.2.2.2. Algoritmo do Protocolo 41
3.2.2.3. Testes e Implementação 46
3.2.2.3.1. Proposta da Implementação 46
3.2.2.4. Lemas e Teoremas 53
3.3. Processos Periódicos e Aperiódicos Independentes (AMB 3) 56
3.3.1. Definição do Problema 56
3.3.2. Servidor Background 56
3.3.2.1. Definição do Servidor Background 56
3.3.3. Servidor Polling 58
3.3.3.1. Definição do Servidor Polling 58
3.3.3.2. Testes e Implementação 60
3.3.4. Servidor DS - Deferrable Server 65
3.3.4.1. Definição do Servidor DS 65
3.3.4.2. Testes e Implementação 68

4. Escalonador Dinâmico "Earliest Deadline" 73
4.1. Processos Periódicos e Independentes (AMB 4) 73
4.2. Processos Periódicos e Dependentes 76
4.2.1. Rendezvous Determinístico (AMB 5) 77
4.2.1.1. Definição do Problema 77
4.2.1.2. Definição da Técnica de "Prazos Revisados" 83
4.2.1.2.1. Algoritmo da Técnica 84
4.2.1.2.2. Análises mais Profundas 87
4.2.1.2.3. Proposta para Garantir a Viabilidade do do Escalonador 93
4.2.1.3. Testes e Implementação 95
4.2.1.4. Lemas e Teoremas 100
4.2.2. Monitor "Kernelized" (AMB 6) 102
4.2.2.1. Definição do Problema 102
4.2.2.2. Modelagem do Monitor 103
4.2.2.3. Definição da Técnica de "Região Proibida" 106
4.2.2.3.1. Algoritmo da Técnica 107
4.2.2.4. Testes e Implementação 115
4.2.2.5. Lemas e Teoremas 117

5. Conclusão 120

5.1. Resultados da Análise 120

5.2. Comparação entre "Earliest Deadline" e "Taxa Monotônica" 121

5.3. Desempenho das Ferramentas Usadas 122

5.4. Sugestões 123

5.5. Considerações Finais 124

Bibliografia

Lista de Figuras

figura 2.1 Classificação de Ambientes para as Análises 12

figura 3.1 Problema de Inversão de Prioridade do Exemplo I 21
figura 3.2 Propriedade de Transitividade do Protocolo de Herança de Prioridade 25
figura 3.3 Funcionamento do Protocolo de Herança de Prioridade 26
figura 3.4 Tratamentos de um Processo durante a Solicitação de Entrada de R.C. - Protocolo de Herança de Prioridade 32
figura 3.5 Tratamentos de um Processo durante a Solicitação de Saída de R.C. - Protocolo de Herança de Prioridade 34
figura 3.6 Problema de "Deadlock" do Exemplo III 38
figura 3.7 Problema de "Cadeia de Bloqueios" do Exemplo IV 39
figura 3.8 Problema de "Deadlock" Resolvido Usando o Protocolo de Prioridade Topo 43
figura 3.9 Problema de "Cadeia de Bloqueis" Resolvido Usando o Protocolo de Prioridade Topo 45
figura 3.10 Ligação entre as Duas Filas Semlock[] e Semaforo[] 48
figura 3.11 Tratamentos de um Processo durante a Solicitação de Entrada de R.C. - Protocolo de Prioridade Topo 52
figura 3.12 Exemplo do Servidor Background 57
figura 3.13 Exemplo do Servidor Polling 59
figura 3.14 Atendimento Temporizado de um Processo Aperiódico pelo Servidor Polling 63
figura 3.15 Exemplo do Servidor DS 67
figura 3.16 Atendimento e Espera de Chegadas Temporizados de um Processo pelo Servidor DS 70

figura 4.1 Escalonador "Earliest Deadline" no Exemplo X 74
figura 4.2 Vencimento de Prazo do Processo A - Exemplo XI 74
figura 4.3 Pontos de Comunicação entre P_A e P_B 78

figura 4.4 Propriedade de Transitividade entre dois Processos Comunicantes 79
figura 4.5 Relação de Precedência entre dois Processos Comunicantes 80
figura 4.6 Comunicação Rendezvous entre Processos do Exemplo XII 82
figura 4.7 Vencimento de Prazo do Processo T_1 do Exemplo XII 82
figura 4.8 Grafo de Relação de Precedência entre os Blocos 85
figura 4.9 Aplicação da Técnica "Prazos Revisados" no Exemplo XIII 86
figura 4.10 Resultado de Escalonamento do Exemplo XIII 87
figura 4.11 Rendezvous entre os Processos do Exemplo XIV 88
figura 4.12 Aplicação da Técnica "Prazos Revisados" no Exemplo XIV 89
figura 4.13 Resultado de Escalonamento do Exemplo XIV 89
figura 4.14 Rendezvous entre Processos do Exemplo XV 90
figura 4.15 Aplicação da Técnica "Prazos Revisados" no Exemplo XV 91
figura 4.16 Resultado de Escalonamento do Exemplo XV 91
figura 4.17 Mudança de Pontos de Comunicação do Exemplo XV 92
figura 4.18 Inclusão de Processos Independentes no Grafo de Execução 94
figura 4.19 Eliminação de "Start Time" de um Processo Periódico 99
figura 4.20 Relacionamento entre Processo, Blocos e Miniblocos 104
figura 4.21 Resultado de Escalonamento do Exemplo XVI 105
figura 4.22 Funcionamento da Técnica "Região Proibida" 106
figura 4.23 Configuração Global dos Processos do Exemplo XVI 109
figura 4.24 Função do Processo de Controle durante Teste 116

Lista de Tabelas

tabela 3.1 Dados dos Processos do Exemplo VII 57
tabela 3.2 Nível de Prioridades dos Processos e sua Descrição Funcional durante Teste doo Servidor Polling 61
tabela 3.3 Principais Diferenças entre Servidor Polling e DS 66
tabela 3.4 Nível de Prioridades dos Processos e sua Descrição Funcional durante Teste do Servidor DS 69

tabela 4.1 Dados dos Processos do Exemplo X 73
tabela 4.2 Dados dos Processos do Exemplo XI 74
tabela 4.3 Dados dos Processos do Exemplo XII 81
tabela 4.4 Dados de Entrada para o Programa REVISA.C 96
tabela 4.5 Dados de Entrada para o Simulador SSE 100
tabela 4.6 Dados dos Processos do Exemplo XVI 105
tabela 4.7 As Regiões Proibidas Calculadas do Exemplo XVI 114

CAPÍTULO 1

INTRODUÇÃO

1. Introdução

1. INTRODUÇÃO

Nos últimos anos a ampliação nas aplicações dos sistemas de tempo real atingiu áreas mais sofisticadas, desde o controle de experiências laboratoriais e de motores automobilísticos, as usinas nucleares, controle de processos, robótica e sistemas-guia de aeronaves. [STA 88]. Geralmente, o computador usado nessas aplicações é constituído de diversos processos críticos (de monitoramento, controle etc) com certas imposições de temporização. Assim, é insuficiente que esses processos sejam corretos apenas do ponto de vista lógico, isto é, não basta uma implementação adequada dos algoritmos.

A grande preocupação no desenvolvimento destes sistemas, denominados Sistemas de Tempo Real Crítico (Hard Real Time System - HTRS), envolve a garantia da funcionalidade dos mesmos sob as limitações de tempo impostas aos processos computacionais, tais como o tempo máximo de resposta ao estímulo externo, as taxas periódicas de atualização das variáveis dos processos sob seu controle, etc [MOK 84].

Uma vez que o atendimento das características de temporização absolutas dos processos é função não apenas da velocidade do processador, mas também da política de compartilhamento de recursos (principalmente da CPU), a eficiência e a confiabilidade do HRTS depende em grande parte de uma estratégia cuidadosa de escalonamento. Assim, o escalonador assume um papel primordial no HRTS dadas as exigências cada vez mais significativas sobre a previsibilidade dos sistemas.

O objetivo deste trabalho está concentrado na análise e estudo dos algoritmos de escalonamento em diversos ambientes de HRTS momoprocessado. A abordagem principal baseia-se em dois algoritmos de escalonamento : estático (Taxa Monotônica) e dinâmico (Earliest Deadline), sendo ambos "Preemptivos" e de "Prioridade Dirigida".

Estes escalonadores confrontam problemas cruciais quando o sistema contém processos periódicos e dependentes e/ou processos aperiódicos, porém, apresentam um desempenho ótimo quando aplicados a um conjunto de processos periódicos e independentes.

Os estudos feitos sobre o escalonador estático concentram-se nos problemas de inversão de prioridades para os processos periódicos dependentes e nos servidores de processos aperiódicos. Já no escalonador dinâmico, a discussão se aprofunda em problemas de troca de mensagens e das regiões compartilhadas para os processos dependentes, os quais não são triviais quando lembramos que cada processo deve cumprir suas restrições de temporização.

Este trabalho está estruturado em cinco capítulos :

Capítulo 1 ⟶ Introdução

Capítulo 2 ⟶ São apresentados os conceitos básicos de HRTS, processos e escalonadores; um diagrama de abordagem das análises realizadas e as ferramentas usadas durante os testes dos algoritmos.

Capítulo 3 ⟶ São apresentados os problemas inerentes ao escalonador estático quando usado em vários tipos de conjuntos de processos e os algoritmos propostos para solucioná-los.

Capítulo 4 ⟶ São levantados os problemas do escalonador dinâmico "Earliest Deadline" quando os processos possuem comunicação mútua e compartilham dados comuns; contém também algoritmos para a solução destes problemas.

Capítulo 5 ⟶ Análises dos resultados, conclusões e sugestões para estudos posteriores.

CAPÍTULO 2 / BASES TEÓRICAS E CONCEITOS FUNDAMENTAIS

2.1. Definição de Hard Real Time System
2.2. Processos em HRTS
2.3. Escalonadores em HRTS
2.4. Diagrama para as análises
2.5. Ferramentas para testes de algoritmos

2. Bases Teóricas e Conceitos Fundamentais

O objetivo deste capítulo é, além de definir HRTS, processos e escalonadores, extrair os ambientes considerados importantes e estruturar seus problemas em duas linhas principais de análise. Um diagrama será apresentado para dar uma visão clara do trabalho e ao mesmo tempo guiar os leitores com relação às análises realizadas.

2.1. Definição de Hard Real Time System - HRTS

Um sistema onde as requisições do usuário podem ser produzidas em um intervalo de tempo razoável e, além disso, pequenas variações de atraso são toleráveis, denomina-se "Soft Real Time System". Quando o mundo externo introduz no sistema computacional um conjunto severo de restrições de temporização contendo, usualmente, um valor de tempo antes do qual o processo não está pronto para o processamento e, um outro valor de tempo, após o qual, caso o processo não tenha terminado a sua execução o sistema poderá sofrer várias consequências, o sistema é caracterizado como um "Hard Real Time System" - HRTS. [MAN 67]

As restrições críticas de temporização no HRTS geralmente são determinadas pelos processos físicos que estão sob seu controle e devem ser encontradas a fim de se evitar possíveis resultados indesejados [MOK 84].

Na verdade, o objetivo central do HRTS é cumprir as requisições de temporização de cada processo e, consequentemente, garantir a previsibilidade do sistema. O comportamento funcional e temporal dos processos deve ser necessariamente determinístico para que possam ser satisfeitas as especificações do sistema. [STA 88]

Processos (ou tarefas) são identidades básicas em um sistema de tempo real; as suas características principais serão apresentadas a seguir.

2.2. Processos em HRTS

2.2.1. Definição

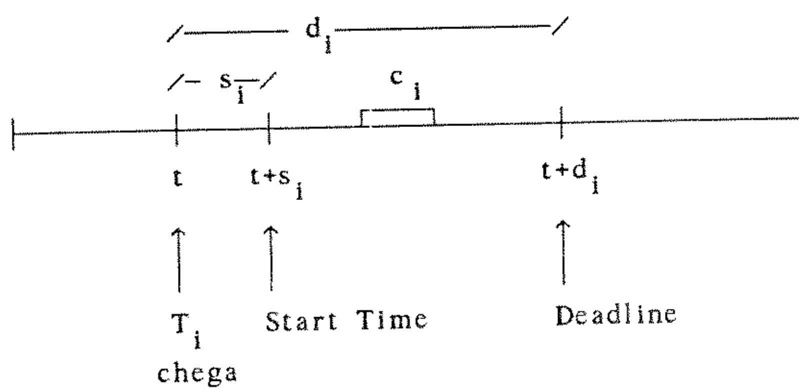
Processo é a abstração de uma sequência de código computacional que, em qualquer ponto da sua execução, pode ser caracterizado pelas informações de estado do processo como, por exemplo, seu contador de programa, sua pilha e os valores das suas variáveis estáticas [MOK 83]. Do ponto de vista do sistema, cada processo

realiza um determinado serviço, contribuindo para o funcionamento integral do sistema.

Os processos em ambientes de HRTS possuem, segundo a definição anterior, certas restrições de temporização as quais são determinadas pelo projetista do sistema de acordo com as características da aplicação. Por exemplo, num sistema de deteção de incêndios, o processo de leitura da temperatura pode ser projetado para ser ativado periodicamente de modo a efetuar a coleta de dados do sensor correspondente (processo físico); ou então, podemos restringir a leitura, a partir do instante 3 e finalizá-la antes do instante 10, etc.

Normalmente, os parâmetros "Start Time" e "Deadline" são considerados as principais requisições de temporização para os processos em HRTS. O "Start Time" impõe um instante de tempo, a partir do qual o processo está na fila de prontos, ou seja, em condições de ser executado; o "Deadline" delimita um prazo antes do qual a execução do processo deve ser concluída.

Quando um processo T_i chega ao sistema no instante t, ele deve estar pronto para ser executado, a partir de seu "Start Time" s_i, valor relativo a t, e deve ser finalizado antes do "Deadline" especificado d_i , também relativo a t.



O processo só é executado dentro do intervalo $[t+s_i, t+d_i]$.

Outro parâmetro importante de um processo é o seu tempo de execução c_i. Este valor, fornecido pelo projetista, geralmente é calculado como o tempo máximo de processamento de um processo em qualquer situação. Uma vez estabelecido o seu valor, ele deve ser suposto fixo e constante, não variando em função do tempo. Assim, sua estimativa deve ser feita com cuidado, de modo a não violar essa condição. O tempo necessário para atender as requisições do processo e para trocar os processos, durante uma mudança de contexto, podem ser incorporados ao seu valor, ou seja, no tempo de execução do processo.

2.2.2. Tipos de Processos

Existem dois tipos de processos : periódicos e aperiódicos.

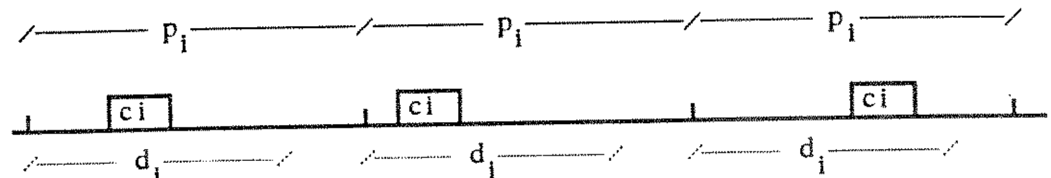
PROCESSOS PERIÓDICOS

Processos periódicos são aqueles cujas execuções são solicitadas em intervalos de tempo regulares, ou seja, suas ocorrências repetem-se a cada período bem definido.

Define-se p_i como sendo o período do processo T_i ; então, diz-se que um processo periódico possui três parâmetros principais :

$$T_i = (c_i , p_i , d_i)$$

onde c_i é o tempo de execução do processo
p_i é o período do processo
d_i é o prazo do processo



Geralmente convenciona-se coincidir o "Start Time" com o instante inicial de cada período. Observa-se, também, que o tempo de execução, c_i, deve ser menor ou igual ao seu prazo d_i, que por sua vez é menor ou igual ao período p_i. Assim sendo, tem-se :

$$c_i \le d_i \le p_i$$

O uso mais comum dos processos periódicos, na prática, é para processar os dados de um sensor e atualizar o estado corrente do sistema de tempo real. Essas aplicações caracterizam o ambiente de HRTS, pois elas impõem um prazo de execução rígido durante o seu uso, já que a execução de um processo tem de terminar antes do início de um novo período.

PROCESSOS APERIÓDICOS

Processos que chegam ao sistema em tempos irregulares, ou seja, em qualquer instante, são denominados processos aperiódicos. Normalmente, esse tipo de processo não possui um prazo rígido para terminar a sua execução; contudo, exige um tempo médio de resposta bastante curto [SPR 89].

Se um processo aperiódico exige a conclusão de sua execução dentro de um prazo rígido, a partir do momento de sua chegada e possui um intervalo mínimo entre suas chegadas, então é chamado processo esporádico [MOK 84], [SPR 89]. Deve ser observado que sem a restrição de tempo mínino entre as chegadas, não seria possível garantir que o prazo do processo esporádico possa ser respeitado, podendo o mesmo monopolizar os recursos do sistema [MOK 84].

2.2.3. Relação entre Processos

Os processsos, do ponto de vista da interação entre si, podem ser classificados em : independentes e dependentes.

Os processos são ditos independentes, se suas execuções não precisam ser sincronizadas, ou seja, a execução de um processo não depende do início ou término das execuções de outros processos [LIU 73]. Os processos criam uma relação de dependência entre si, quando eles realizam uma troca de mensagens ou compartilham dados comuns.

A troca de mensagens geralmente é conseguida através das primitivas de enviar (SEND) e receber (RECEIVE) informações. Se um processo envia uma mensagem para outro processo e exige a confirmação do recebimento desta, então o ato é chamado síncrono; caso simplesmente ele envie mensagem e continue a sua execução sem precisar ter a confirmação de que a mensagem foi recebida, então trata-se de um ato assíncrono. Portanto, a relação entre os processos está ligada à forma de comunicação entre eles, pois a necessidade de espera para o recebimento e emissão de mensagens cria uma ligação involuntária entre os processos na qual a execução de um depende da execução do outro (processos síncronos). Neste caso, uma relação de precedência também é incorporada neles, pois a ordem de execução dos processos, a que está ligada a liberação e o bloqueio dos processos, é imposta e deve ser respeitada.

Para organizar o uso de dados comuns normalmente é aplicado entre os processos as regiões críticas, guardadas pelo mesmo semáforo, e dentro delas só é permitida a execução de um único processo de cada vez, ou seja, quando um processo estiver na sua região crítica outros processos que solicitarem a entrada em suas

regiões críticas deverão ser barrados, sendo necessário esperar o término do primeiro para que seja possível executar o próximo. Assim, a dependência interprocessos é formada. Porém, o compartilhamento de dados comuns não constitui a relação de precedência interprocessos e sim apenas a dependência do término de execução na região crítica de um em relação a outro.

2.3. ESCALONADORES EM HRTS

2.3.1. DEFINIÇÃO E CARACTERÍSTICAS

A função do escalonador é especificar uma ordem na qual todos os processos no sistema devem ser executados, de modo que satisfaçam suas restrições de temporização (prazo, "start time", etc) impostas pela aplicação [DHA 78].

Para alcançar um melhor desempenho do escalonador, existem vários algoritmos de escalonamento, cada um dos quais fornecendo um conjunto de regras para determinar qual processo será executado em um determinado momento [DHA 78]. Uma maneira simples de especificar o algoritmo de escalonamento é atribuir prioridades aos processos tal que os de prioridade alta exercem uma precedência de execução sobre os de prioridade baixa.

Assim, uma especificação de tais algoritmos equivale a uma especificação do método de determinação das prioridades dos processos. Geralmente, esse tipo de escalonador é chamado de "Prioridade Dirigida" [DHA 78].

Outra estratégia importante usada pelo escalonador é a "preempção". Um escalonador preemptivo permite a interrupção de um processo em execução, suspendendo-o, salvando o estado atual do processo e alocando a CPU a um outro processo. Já o não-preemptivo é aquele que aloca a CPU a um processo até o término de sua execução, sem interrupção sob qualquer motivo. O último leva a vantagem sobre o primeiro pelo fato de gastar menos tempo na manipulação dos processos. No entanto, sua inflexibilidade não favorece a elaboração de algoritmos de escalonamento ótimos.

Portanto, escolheu-se o algoritmo de escalonamento "Prioridade Dirigida e Preemptivo" como ambiente central, a partir do qual uma série de análises serão realizadas.

Define-se um "algoritmo de escalonamento viável" (ou praticável) como sendo um algoritmo, segundo o qual as restrições de temporização (prazo, etc) de todos os processos são cumpridas [DHA 78]. Por outro lado, se as execuções de alguns

processos não forem completadas, antes dos seus prazos, então o algoritmo que escalona esse conjunto de processos é dito inviável.

2.3.2. Ambientes de Monoprocessamento, Multiprocessamento e Sistemas Distribuíd

MONOPROCESSAMENTO

Pelo fato dos sistemas de monoprocessamento terem sido os primeiros sistemas desenvolvidos, eles constituíram o campo de pesquisa pioneira na área de escalonadores em HRTS, o que resultou num grande número de trabalhos nessa direção. Manacher [MAN 67] derivou um algoritmo para a geração de escalonadores em HRTS, porém seus resultados estão restritos a algumas situações irreais, onde há somente um único tempo de requisição (tempo em que o processo ficar pronto) para todos os processos. Lampson [LAM 68] discutiu o problema de escalonamento em geral e apresentou um conjunto de procedimentos de multiprogramação, em ALGOL, os quais podem ser implementados em Software ou projetados como uma forma de escalonador especial. Contudo, o trabalho de Liu e Layland [LIU 73] destacou-se por mostrar resultados imprescindíveis para o escalonamento nos sistemas de HRTS com monoprocessamento como, por exemplo, a determinação de escalonadores ótimos para sistemas monoprocessadores.

Escolheu-se o sistema de monoprocessamento para suportar o ambiente central de análises deste trabalho por ser o mais simples, considerando-se a análise de certos problemas tais como a dependência entre processos, etc.

MULTIPROCESSAMENTO

Conseguir um algoritmo de escalonamento ótimo em um ambiente de multiprocessamento é um problema NP_Hard, ou seja, computacionalmente intratável. Mok e Dertouzos [MOK 78] provaram que para os sistemas de multiprocessamento não existe algoritmo ótimo para escalonar processos sem ter conhecimento a priori de prazos, tempos de execução e "start time" dos processos.

SISTEMAS DISTRIBUÍDOS

Atualmente os sistemas distribuídos têm tido uma atuação significativa nas aplicações de HRTS.

Como característica de um HRTS, um bom algoritmo de escalonamento neste sistema é aquele que garante um número máximo de processos, satisfazendo seus prazos correspondentes. Porém, a dificuldade adicional do escalonador em sistemas distribuídos, em relação aos mono e multiprocessamento, são os problemas introduzidos

devido aos atrasos de comunicação entre os nós do sistema [RAM 89].

Normalmente, o escalonador em sistemas distribuídos consiste em dois componentes principais : um local e outro distribuído. Quando um processo chega a um nó do sistema, o escalonador local decide a garantia do processo. Aqui o termo "garantia de um processo" se refere à situação na qual o processo cumprirá sua execução antes do vencimento do prazo e, ainda, a sua inclusão não afetará os processos já garantidos anteriormente no mesmo nó [RAM 89]. Caso o processo não seja garantido localmente, então todos os componentes de escalonamento locais cooperam entre si a fim de escolher um nó que possua recursos suficientes para garantir este processo. Se tal nó existe, então o processo é enviado imediatamente ao nó eleito; caso contrário, o processo é rejeitado pelo sistema [RAM 89].

Existem vários métodos para selecionar esse nó ao qual o processo que não foi garantido localmente é enviado. Uma maneira simples, chamada "algoritmo de escalonamento randômico", envia o processo a um outro nó aleatoriamente na tentativa de encontrar um nó adequado que garanta o processo. Dois métodos cooperativos que usam as informações de estado do sistema são propostas por Stankovic et al [RAM 89] : **Bidding** e **Focus Addressing**. Em "focus addressing" o nó com a estimativa de disponibilidade de recursos alta é selecionado; em "bidding", o nó é escolhido se ele fizer a melhor oferta.

O custo de comunicação envolvido no "bidding" é alto, porém sua seleção baseia-se em informações relativamente confiáveis e seguras. Já o "focus addressing" realiza menos comunicação, mas suas informações de estado, incompletas e inseguras, podem levar a decisões erradas [RAM 89].

2.3.3. O Algoritmo de Escalonamento "Prioridade Dirigida" e "Preemptivo"

Nesse algoritmo de escalonamento, os processos são dotados de prioridades, e a execução dos processos de prioridade baixa será interrompida, quando os de prioridade maior solicitam a execução, mesmo sem ter finalizado a sua execução. Porém, a execução interrompida deve ser recuperada posteriormente. Pode ser assumido que o tempo gasto pela troca de execução dos processos é nulo, mas essa suposição não é realista. De qualquer forma, desde que o tempo de execução de um processo seja interpretado como ser o tempo máximo de seu processamento, então o tempo gasto com a troca de processos durante a ocorrência de preempção também pode ser incluido dentro do tempo de execução dos processos [DHA 78].

Os algoritmos de escalonamento "Prioridade Dirigida e Preemptivo"

podem ser classificados em duas categorias : escalonador estático ou de prioridade pré-fixada, e escalonador dinâmico.

2.3.3.1. Escalonadores Estáticos

Também conhecidos como algoritmos de prioridade pré-fixada. As prioridades dos processos são determinadas a priori e não mudam de valor no decorrer do sistema [LIU 73]. Ou melhor, na inicialização do sistema, a cada processo é atribuído uma prioridade fixa e o processo será executado sempre de acordo com esse valor de prioridade.

Às vezes, o critério para determinar-se as prioridades é aleatório, dependendo da intuição ou do conhecimento sobre os processos de quem projeta o sistema. Porém, existe uma regra para determinar as prioridades, a qual atribui a maior prioridade ao processo de período mais curto, e a menor prioridade ao processo de período mais longo. Isso parece razoável, se os processos que ocorrem mais frequentemente no sistema apresentarem maior urgência de serem executados. Essa maneira de pré-fixar as prioridades é chamada "Taxa Monotônica".

2.3.3.2. Escalonadores Dinâmicos

O escalonador dinâmico renova as prioridades dos processos, durante a vida do sistema, ou seja, a cada ocorrência de um mesmo processo é determinado uma prioridade, possivelmente diferente.

Existem dois algoritmos de escalonamento dinâmico que apresentam um ótimo desempenho quando o ambiente é de monoprocessamento. São eles : "Earliest Deadline" e "Least Slack".

O algoritmo de escalonamento "Earliest Deadline" ou "Deadline Dirigido" atribui a prioridade de acordo com a proximidade do prazo da atual instância do processo [LIU 73]. O processo recebe prioridade alta se o prazo ("deadline") da ocorrência atual é mais próximo, e baixa prioridade, se o prazo referido é mais distante. Assim, a qualquer instante, um processo com prioridade mais alta, e que ainda não executou, pode ser escalonado como o próximo a ser executado.

O algoritmo "Least Slack " atribui alta prioridade ao processo que possui a "folga" menor [MOK 84]; o valor de "folga" é calculado como o tempo existente para o vencimento do prazo atual do processo, descontando o tempo de execução restante para sua conclusão.

Sejam $cs_i(t)$, o resto do tempo de execução de um processo T_i; t é o instante atual e $d_i(t)$, seu prazo atual, então define-se "folga" do processo T_i como o valor $(d_i(t)-t-cs_i(t))$. No outro sentido, "folga" representa o tempo máximo pelo qual o escalonador pode adiar a execução do processo. Este algoritmo é ótimo [MOK 83], assim como o "earliest Deadline", quando os processos são independentes e periódicos num sistema de monoprocessamento.

2.4. Diagrama para as Análises do Trabalho

Nos itens anteriores foram discutidos os conceitos básicos sobre o HRTS, processos e escalonadores. Será construído agora um diagrama de vários ambientes sobre o qual esse trabalho desenvolverá uma série de análises e estudos. Particularmente, serão escolhidos para uma análise mais aprofundada os algoritmos "Taxa Monotônica" como representante dos escalonadores estáticos e "Earliest Deadline" como representante dos escalonadores dinâmicos. O diagrama na figura 2.1 mostra uma hierarquia para classificação desses ambientes.

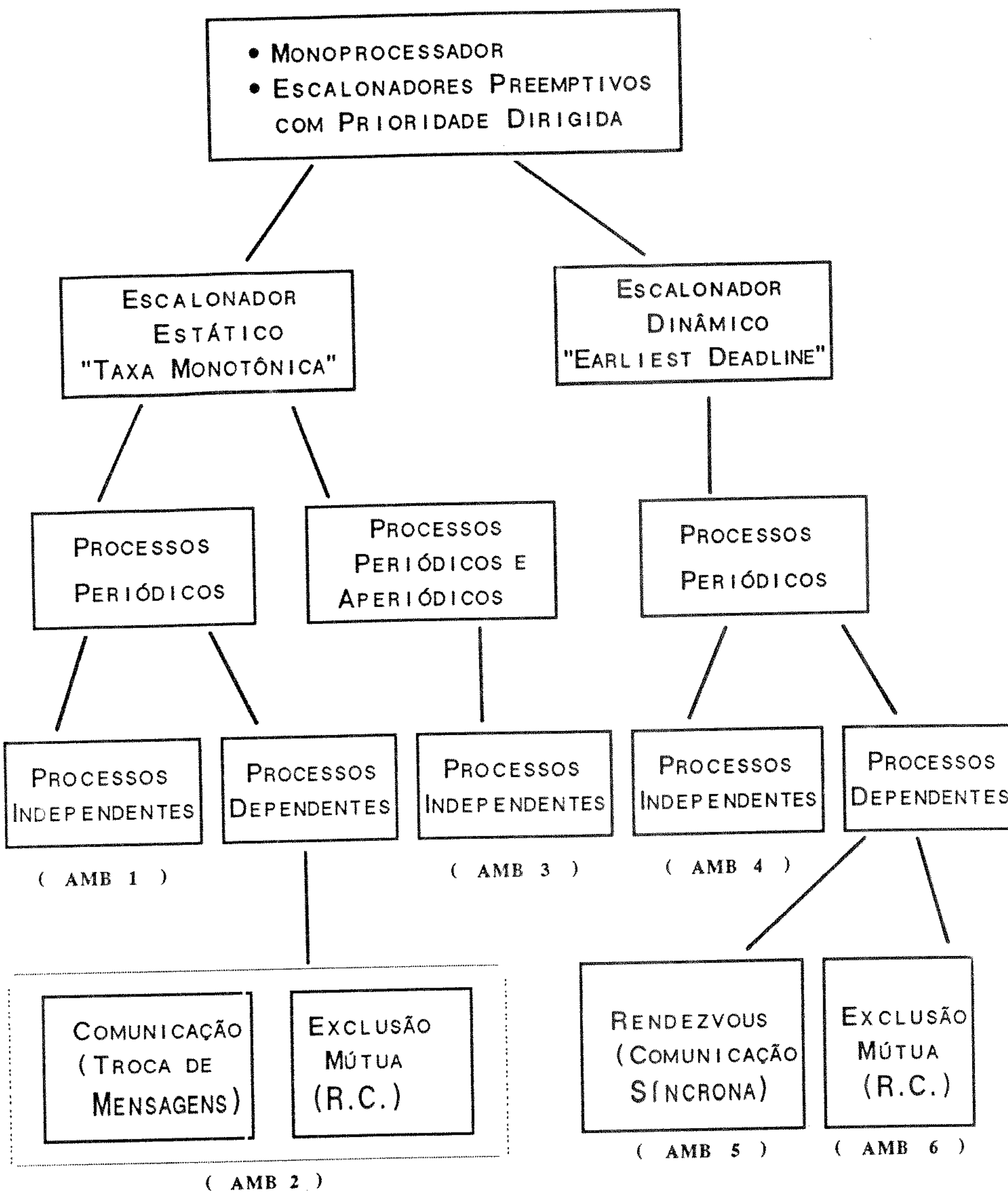


figura 2.1 - Classificação de Ambientes para as Análises

Cada bloco na figura contém uma condição característica sobre o ambiente, sendo que os blocos superiores apresentam uma condição mais abrangente e os de nível inferior são suas subclassificações. Assim, os blocos terminais atingem classificações bem específicas, ou seja, os vários ambientes para as análises.

A periodicidade e a questão de dependência entre os processos são dois grandes parâmetros usados na análise. Os processos podem ser periódicos ou aperiódicos, dependentes ou independentes. Porém, os sistemas que contêm apenas processos aperiódicos não serão incluídos nessa discussão, pois a não previsibilidade das chegadas dos processos torna o escalonamento impraticável.

Assim, podemos observar seis ambientes de sistema de monoprocessamento com escalonadores tipo "Prioridade Dirigida e Preemptivo" :

AMB1) Algoritmo de escalonamento estático (prioridade pré-fixada) "Taxa Monotônica" com processos periódicos e independentes

AMB2) Algoritmo de escalonamento estático (prioridade pré-fixada) "Taxa Monotônica" com processos dependentes.

AMB3) Algoritmo de escalonamento estático (prioridade pré-fixada) "Taxa Monotônica" com processos periódicos e aperiódicos, ambos independentes

AMB4) Algoritmo de escalonamento dinâmico "Earliest Deadline" com processos independentes e periódicos

AMB5) Algoritmo de escalonamento dinâmico "Earliest Deadline" com processos periódicos e dependentes (via comunicação síncrona)

AMB6) Algoritmo de escalonamento dinâmico "Earliest Deadline" com processos periódicos e dependentes por compartilhamento de dados comuns

Deve-se reiterar que este trabalho baseia-se somente nos sistemas de monoprocessamento.

O capítulo 3 será responsável pelas análises do escalonador de prioridade pré-fixada, englobando os ambientes AMB1, AMB2 e AMB3.

O capítulo 4 tratará o escalonador "Earliest Deadline", encarregando-se das discussões sobre os ambientes AMB4, AMB5, AMB6.

2.5. Ferramentas para Testes de Algoritmos

Durante as análises dos algoritmos de escalonamento, os simuladores, núcleos e softwares de suporte são fundamentais para validar e testar

esses algoritmos estudados.

Porém, é difícil escolher uma ferramenta que seja ao mesmo tempo adequada e de fácil utilização para todas as análises propostas no trabalho; assim, durante o início deste trabalho não encontraram-se muitas opções de ambientes de testes disponíveis. Inicialmente tentou-se o uso do software de suporte STER, desenvolvido por Coello [COE 86]. Logo, certas dificuldades se evidenciaram no teste de algoritmos em ambientes de escalonamento dinâmico. Assim sendo, optou-se pelo núcleo de tempo real DEADMOSI, que foi elaborado após o início deste trabalho, pela flexibilidade funcional do seu escalonador. Porém, do ponto de vista dos testes ele apresenta certas inconveniências, tanto em relação ao tempo gasto para os testes quanto à manipulação do próprio núcleo.

Tendo em vista esses problemas, surgiu a idéia de se elaborar um simulador SSE para facilitar estudos mais aprofundados sobre escalonadores em HRTS para o grupo de pesquisa da área de sistemas de tempo real do departamento DCA da Engenharia Elétrica da UNICAMP. Apesar do SSE não ter sido muito usado neste trabalho a sua importância é inegável e portanto merece uma descrição sucinta que será apresentada no item 2.5.2.

2.5.1. Núcleo de Tempo Real DEADMOSI

Poucos núcleos existentes na área de tempo real trazem considerações quanto à restrição de temporização de cada processo no sistema; o núcleo DEADMOSI foi desenvolvido, por Azevedo [AZE 91], justamente devido à necessidade dessa nova abordagem no campo de pesquisa e desenvolvimento de sistemas de tempo real.

Sua interface com o usuário segue um padrão internacional - Norma MOSI, viabilizando o transporte do aplicativo para outros ambientes.

Na verdade, foram implementadas três políticas de escalonamento diferentes, a fim de oferecer aos usuários várias técnicas de análise de escalonamento, são eles QUASIMOSI, DEADMOSI e LAXIMOSI.

O QUASIMOSI adota a política de escalonamento estática, ou seja, pré-fixa as prioridades de todos os processos no sistema. O algoritmo de escalonamento "Earliest Deadline" está incorporado ao núcleo DEADMOSI, cujas funções se enquadram no contexto de análises deste trabalho. O terceiro núcleo, denominado LAXIMOSI, implementa a estratégia "Least Laxity" (ou "Least Slack") de escalonar processos, a qual atribui dinamicamente as prioridades aos processos segundo suas folgas.

Atualmente, esses núcleos estão sendo utilizados no Instituto de

Automação do Centro Tecnológico para Informática (CTI).

2.5.1.1. Características Gerais do DEADMOSI

Como já foi citado, o núcleo foi estruturado segundo a norma MOSI, a qual fornece uma convenção de interface para aumentar a portabilidade dos aplicativos, quando os serviços acessados nestes são transportados entre os diversos ambientes operacionais.

O padrão MOSI agrupa as funções, de acordo com seus tipos de serviços, chamados "capabilities", e o núcleo DEADMOSI implementou a maioria deles [AZE 89] :

1 - *Memory Management* ⟶ Gerencia a memória responsabilizando-se pela aquisição e liberação de blocos da memória.

2 - *Time Management* ⟶ Controla as temporizações dos processos, usando funções tais como DELAY, SLEEP, etc.

3 - *Process Management* ⟶ Define as funções que suportam a criação e o controle de processos periódicos e aperiódicos.

4 - *Process Syncronization* ⟶ É responsável pelas manipulações de sincronização e comunicação entre os processos utilizando-se de semáforos e de mecanismos de troca de mensagens.

5 - *Exception Handling* ⟶ Define as funções que habilitam o tratamento de exceções. Estas são eventos que necessitam de um tratamento que interrompa o ciclo normal de execução.

O núcleo DEADMOSI reconhece três tipos de processos :

(1) *Processos periódicos com prazos definidos*

(2) *Processos aperiódicos com prazos definidos*

(3) *Processos aperiódicos com prioridades definidas*

É fácil perceber que os dois primeiros são tratados com prioridades dinâmicas no decorrer do sistema e o último, com prioridades pré-fixadas.

O núcleo, durante o escalonamento, organiza os processos reconhecidos em dois níveis de prioridade. No primeiro nível, os processos com prazos definidos (podem ser periódicos ou aperiódicos) são ordenados de acordo com o prazo de cada um e, no segundo nível, os processos aperiódicos com prioridades pré-definidas são organizados segundo suas prioridades. Sendo assim, o núcleo escalona primeiro os

processos do primeiro nível e, depois, se nenhum processo estiver pendente no primeiro nível, os do segundo nível são executados.

Uma função importante fornecida pelo núcleo é a verificação, feita em cada "tick" de tempo, de "estouro" do prazo de algum processo no sistema. Assim que o núcleo detectar um estouro, a rotina **"ostrdead()"**, responsável pelo tratamento da situação ocorrida, deve ser chamada pelo usuário para uma verificação mais clara do acontecimento.

Como as aplicações do núcleo são bem diversificadas, cada uma das quais possuindo características particulares, um configurador de núcleo foi criado para direcionar as necessidades do aplicativo do usuário de acordo com o seu tipo. Isto é possibilitado somente devido à característica do núcleo de explorar a modularidade da implementação. A interface do configurador com o usuário dispensa o conhecimento dos elementos internos do núcleo para montar ou alterar as estruturas do mesmo.

O configurador gera uma biblioteca que será utilizada na implementação do programa de aplicação.Na verdade, os processos do aplicativo devem ser codificados separadamente em linguagem C , podendo então usar os serviços fornecidos pelo núcleo, uma vez providenciada a inicialização dos processos e as declarações necessárias. Feito isso, o programa contendo os processos é compilado e "linkado" junto com a biblioteca criada pelo configurador.

2.5.2. Simulador SSE (Sistema de Simulação do Escalonador)

O simulador SSE foi desenvolvido, por Spanó [SPA 91], devido à necessidade de testes de validação das diversas políticas de escalonamento em ambientes de sistema de tempo real crítico (Hard Real Time System) onde os requisitos de temporização dos processos devem ser respeitados.

O SSE oferece aos usuários, não apenas as facilidades para análise dos aspectos cruciais dos escalonadores, mas, também, uma interface de operação amigável. Sem dúvida, ele representa uma ferramenta importante para o grupo de trabalho relacionado à área. Visto que sua conclusão só ocorreu na fase final deste trabalho, seu uso nas análises do mesmo não foi extenso; contudo, seu auxílio tem sido de grande valor.

2.5.2.1. Características Gerais do SSE

A interface do software simulador com o usuário está baseada no conceito de janelas, permitindo um visual claro e manuseio simples do mesmo.

O SSE possui duas funções principais :

(F1) Serve como ferramenta para as análises de escalonadores, ou seja, o usuário pode testar vários conjuntos de processos usando os escalonadores já incorporados no software. Atualmente, existem quatro tipos de escalonadores instalados: "Earliest Deadline", "Taxa Monotônica", "Least Laxity" e "Servidor DS".

(F2) Caso o usuário deseje implementar e testar um escalonador que não seja nenhum destes, o SSE demonstra sua versatilidade oferecendo um conjunto de comandos-função, de modo que se possa construir um novo escalonador. Nesse caso, o novo escalonador é incorporado ao SSE e as análises podem ser feitas também sobre este.

Quanto à natureza dos processos, o SSE suporta manipulações de processos periódicos, aperiódicos, independentes e os que se comunicam entre si. O ambiente em que o SSE atua, no momento, é o de monoprocessamento.

Geralmente, um simulador difere de um núcleo real em função dos tempos de teste, pois o primeiro não executa processos reais, mas apenas simula o tempo de execução através de eventos, o que economiza um vasto tempo para os usuários, quando o objetivo deste é o de apenas validar os algoritmos. Portanto, simuladores são recomendados nos estudos e nas análises dos algoritmos de escalonamento.

CAPÍTULO 3

ESCALONADOR ESTÁTICO "TAXA MONOTÔNICA"

3.1. Processos periódicos e independentes - AMB 1
3.2. Processos periódicos e dependentes - AMB 2
3.3. Processos periódicos e aperiódicos independentes - AMB 3

3. Escalonador Estático "Taxa Monotônica"

Como foi definido no capítulo anterior, o algoritmo do escalonador estático baseia-se na idéia do estabelecimento das prioridades dos processos no início do processamento do sistema, mantendo-as constantes durante o mesmo. Apesar de levar desvantagem quanto à flexibilidade em relação ao escalonador dinâmico, ele obviamente apresenta menor "overhead".

Lembrando que o escalonador escolhido neste trabalho possui as características de "Prioridade Dirigida" e de "Preempção", a qualquer instante em que um processo ficar pronto ou concluir sua execução o escalonador verifica as prioridades dos processos na fila de prontos e aloca a CPU àquele de maior prioridade.

Este capítulo mostrará vários aspectos e problemas do escalonador estático "Taxa Monotônica" aplicado nos ambientes AMB1, AMB2 e AMB3 (figura 2.1), com destaque a certas propostas pelas quais os algoritmos, tais como os servidores Polling e DS e os protocolos de herança de prioridade e de prioridade topo, podem ser aplicados, a nível de usuário, num núcleo convencional, sem a necessidade de grandes alterações. Estas caracterizam um apoio ao usuário quando as modificações sobre o núcleo (ou sobre o escalonador) são impossíveis.

3.1. Processos Periódicos e Independentes (AMB 1)

O problema de escalonamento em HRTS torna-se menos complicado quando se trata de processos independentes e periódicos. Dispensando as preocupações com as questões de dependência, o problema resume-se somente à alocação de execução de cada processo do sistema, de modo que não percam seus prazos.

Segundo Liu [LIU 73], o algoritmo "Taxa Monotônica" é ótimo para esse tipo de processos, uma vez que não existem outros algoritmos de prioridade pré-fixada que possam escalonar um conjunto de processos (periódicos e independentes), que por sua vez não possa ser escalonado também pelo algoritmo "Taxa Monotônica". Se existem um ou mais algoritmos praticáveis de atribuição de prioridades para um determinado conjunto de processos, o método "Taxa Monotônica" está entre eles.

Define-se o fator de utilização do processador como sendo a fração de tempo usado pelo processador na execução de processos; em outras palavras, é igual a 1 menos a fração do tempo ocioso do processador durante a atividade do sistema.

Seja (c_i/p_i) a fração de tempo usado pelo processador na execução

do processo T_i , o fator de utilização U do processador para m processos seria :

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

c_i : tempo de execução do processo T_i
p_i : período do processo T_i

Em relação a um algoritmo de atribuição de prioridades fixas, um conjunto de processos é dito "Utilizar Completamente" o processador se, e somente se, este algoritmo é praticável para eles na presente condição e, qualquer aumento no tempo de execução de um dos processos pode resultar na impraticabilidade do algoritmo. Assim, o menor limite superior do fator de utilização do processador U_{MS} é definido como sendo o mínimo dos fatores de utilização de todos os conjuntos que utilizam completamente o processador.

Liu [LIU 73] provou que, para um conjunto de m processos com prioridades pré-fixadas, o menor limite superior de utilização do processador U_{MS} é

$$U_{MS} = m\ (2^{1/m} - 1)\ ;$$

logo, para um conjunto de dois processos $U_{MS} \approx 0.83$ e para o de três processos $U_{MS} \approx 0.78$ e assim por diante.

Se um conjunto de processos possuir seu fator de utilização abaixo do valor de U_{MS} , então deve existir para ele um algoritmo de escalonamento praticável de prioridade pré-fixada. Por outro lado, para aqueles que têm o fator de utilização acima de U_{MS} a praticabilidade do escalonamento só é conseguida se os processos do conjunto são relacionados adequadamente.

Portanto, pode-se obter uma condição suficiente que caracteriza a escalonabilidade do algoritmo "Taxa Monotônica", que é ótimo entre os outros escalonadores estáticos, de um dado conjunto de processos periódicos e independentes.

Assim, um conjunto de n processos escalonados pelo algoritmo de "Taxa Monotônica" pode cumprir sempre os prazos de cada processo se

$$\sum_{i=1}^{n} (c_i/p_i) \leq n \cdot (2^{1/n} - 1)$$

3.2. Processos Periódicos e dependentes (AMB 2)

3.2.1. Problema da Inversão de Prioridade

Um importante problema no contexto dos sistemas de tempo real é o efeito dos bloqueios causados pela necessidade de sincronização dos processos que compartilham recursos lógicos ou físicos.

A aplicação direta de mecanismos de sincronização tais como rendezvous, semáforos, monitores, etc, pode conduzir a uma situação incontrolável denominada "inversão de prioridades" onde um processo de maior prioridade é bloqueado pelo processo de menor prioridade durante um tempo indefinido; em outras palavras, este fenômeno acontece quando o processo de maior prioridade espera a execução do outro, de prioridade menor. O exemplo mais frequente é quando dois processos tentam acessar um dado comum e o processo de menor prioridade chega primeiro, ganhando o acesso ao dado compartilhado; assim, os outros processos de prioridade mais alta serão obrigados a esperar a liberação do recurso.

Aqui, os termos "preempção" e "bloqueio" merecem ser explicados em termos mais claros. "Preempção" é um ato onde um processo de menor prioridade cede a CPU para um processo de maior prioridade; o estado "bloqueado" é uma situação na qual um processo de maior prioridade é forçado a esperar pelo processamento de um outro processo de menor prioridade [SHA 87].

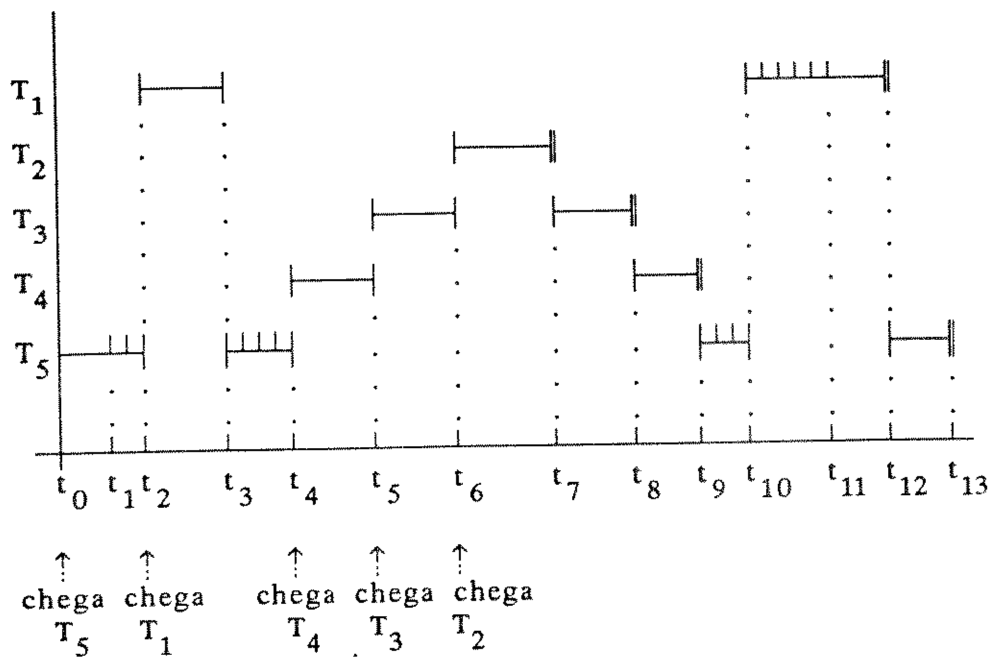
O problema da inversão de prioridade afeta não apenas o grau de escalonabilidade mas também a previsibilidade do sistema, pois o tempo de bloqueio compromete as restrições de temporização (tais como prazo, período, start-time, etc) dos processos. Portanto, serão analisados alguns protocolos que minimizam o número de bloqueios e seus desempenhos no sistema de tempo real, procurando manter um alto grau de escalonabilidade.

Antes de descrever o protocolo, serão vistos alguns exemplos que mostram o problema da inversão de prioridade.

(Exemplo I)

Sejam T_1, T_2, T_3, T_4 e T_5 os processos periódicos com suas prioridades atribuídas em ordem decrescente, isto é, $prio(T_1) > prio(T_2) > prio(T_3) > prio(T_4) > prio(T_5)$. Os processos T_1 e T_5 devem compartilhar a mesma estrutura de

dados guardada pelo semáforo binário S. Suponha que T_5, T_1, T_4, T_3 e T_2 chegam nos instantes t_0, t_2, t_4, t_5 e t_6 , respectivamente. A figura 3.1 mostra a situação referida.



|-|-|-|-|-| ⟶ execução da região crítica guardada pelo semáforo S

|———| ⟶ execução do processo fora da região crítica

figura 3.1 - Problema de Inversão de Prioridade do Exemplo I

t_0 : T_5 chega e executa

t_1 : T_5 requisita a entrada na região crítica guardada pelo semáforo S, logo, T_5 executa a região crítica.

t_2 : T_1 chega e faz preempção de T_5 pois prio(T_1) é maior que prio(T_5). T_1 executa.

t_3 : T_1 requisita e falha na requisição de entrar na região crítica. T_1 é bloqueado por T_5. T_5 então retoma a sua execução.

t_4 : T_4 chega e faz preempção em T_5 pois prio(T_4) é maior que prio(T_5). T_4 executa.

t_5 : T_3 chega e faz preempção em T_4 pois prio(T_3) é maior que prio(T_4) . T_3 executa.

t_6 : T_2 chega e faz preempção em T_3 pois prio(T_2) é maior que prio(T_3). T_2 executa.

t_7 : T_2 finaliza a execução e T_3 retoma a execução. Neste momento T_3 é de maior

prioridade e T_1 ainda está bloqueado.

t_8 : T_3 finaliza a sua execução e T_4 retoma a execução.

t_9 : T_4 finaliza a execução e T_5 retoma a execução.

t_{10} : T_5 sai da região crítica e libera T_1. Agora, T_1 entra na região crítica guardada pelo semáforo S.

t_{11} : T_1 sai da região crítica e continua a execução pois ele é de maior prioridade.

t_{12} : T_1 finaliza a execução e T_5 retoma a execução.

t_{13} : T_5 finaliza a execução.

Observa-se que durante a execução da região crítica pelo processo T_5, o processo de prioridade mais alta T_1 chega e tenta posteriormente o uso de um dado comum. Porém, T_1 foi bloqueado em t_3 pelo semáforo S. Assim, o processo T_1, mesmo sendo de maior prioridade, acabou bloqueado por um tempo maior que o necessário para a execução da região crítica completa por T_5 (de t_3 até t_{10}). Isto se deve às chegadas dos processos de prioridade intermediária, entre as execuções de T_1 e T_5 , durante o bloqueio de T_1 por T_5, pois todos eles, T_2,T_3 e T_4, exercem preempção em T_5 e T_1 deve esperar até que eles completem suas execuções. Logo, T_1 é atrasado indiretamente pelos processos de prioridade mais baixa. Caso o número de processos de prioridade intermediária fosse grande, então o atraso de T_1 seria bem mais longo. De qualquer modo, a duração do bloqueio é, de fato, imprevisível.

Essa situação seria resolvida parcialmente, se no instante t_4 , T_5 não sofresse preempção por parte de T_4, pois existe um processo T_1 de prioridade maior que T_4 que se encontra bloqueado à espera da liberação da região crítica por T_5.

O problema da inversão de prioridades é idêntico no uso de monitores. No caso do mecanismo de rendezvous, quando um processo de prioridade mais alta estiver esperando a primitiva de rendezvous correspondente de um processo de prioridade mais baixa e chegar um terceiro processo de prioridade intermediária, que não se comunica com nenhum dos dois processos, então o processo de prioridade mais baixa pode sofrer preempção por este. Assim, a situação não seria muito diferente daquela que foi discutida anteriormente.

O protocolo de herança de prioridade é um meio de retificar o problema da inversão de prioridade inerente às primitivas de sincronização.

Contudo, antes de se investigar o protocolo propriamente dito, algumas notações e suposições serão definidas e declaradas.

Assume-se que T_1, T_2 ,,, T_n são listados em ordem decrescente de

prioridade, sendo T_1 o processo de maior prioridade entre estes. A cada processo é atribuído uma prioridade pré-fixada. A estrutura de dados compartilhada é sempre guardada por semáforos binários. Usa-se os símbolos $p(s_i)$ e $v(s_i)$ para denotar as operações *wait* e *signal*, respectivamente, do semáforo binário s_i ; $prio(T_i)$ representa a prioridade do processo T_i. A notação $z_{i,j,k}$ tem como significado " a k-ésima ocorrência de uma região crítica do processo T_i guardada pelo semáforo s_j ". Esta notação é necessária pois o processo T_i pode acessar o semáforo s_j mais que uma vez. $z_{i,j}$ denota qualquer ocorrência de qualquer região crítica guardada pelo semáforo s_j no processo T_i e z_i , qualquer região crítica do processo T_i. Um processo pode ter regiões críticas múltiplas que não se sobrepõem, por exemplo :

{ $p(s_1)$ $v(s_1)$ $p(s_2)$ $v(s_2)$ }

E a região crítica pode ser um aninhamento de semáforos, por exemplo :

{ $p(s_1)$ $p(s_2)$ $v(s_2)$ $v(s_1)$ }

A frase "a duração de uma região crítica" refere-se ao tempo de execução total do par *wait-signal* correspondente, ou seja, o tempo de execução que começa de $p(s_1)$ e termina em $v(s_1)$ ou que começa em $p(s_2)$ e termina em $v(s_2)$

3.2.1.1. Definição do Protocolo de Herança de Prioridade

O protocolo está baseado na idéia de que quando um processo J bloqueia processos de prioridade mais alta, ele executará a sua região crítica com a mesma prioridade do processo bloqueado por ele. E depois de sair da região crítica, o processo J retoma sua prioridade original (pré-fixada).

3.2.1.2. Algoritmo do Protocolo

■■ Quando um processo T_i quer acessar a região crítica guardada pelo semáforo s, então :

passo1 -- Antes de entrar na região crítica verifica se s já não está fechado por outro processo; em outras palavras, se algum processo está dentro da região crítica. Caso s ainda não tenha sido fechado então vai para passo3, caso contrário vai para passo2.

passo2 -- O semáforo já está fechado, então o processo T_i é dito bloqueado pelo processo T_j ,o qual fechou o semáforo s. T_j herda a prioridade de T_i .Fim.

passo3 -- O semáforo ainda não foi fechado, então T_i fecha s e entra na região crítica. Fim.

■■ Quando um processo T_i sai da região crítica guardada por s :

passo1 -- Abre o semáforo associado s

passo2 -- Libera os processos bloqueados pela região crítica se houver algum.

passo3 -- Se T_i tinha herdado a prioridade de algum processo de prioridade mais alta, então retoma sua prioridade original.

■■ Quando um processo T_i chega e um outro processo T_j está sendo executado :

passo1 -- Verifica se a prioridade de T_i é mais alta que a prioridade herdada ou original de T_j com a qual este está executando. Caso afirmativo, vai para passo2, senão vai para passo3.

passo2 -- O processo T_i faz preempção em T_j. Fim.

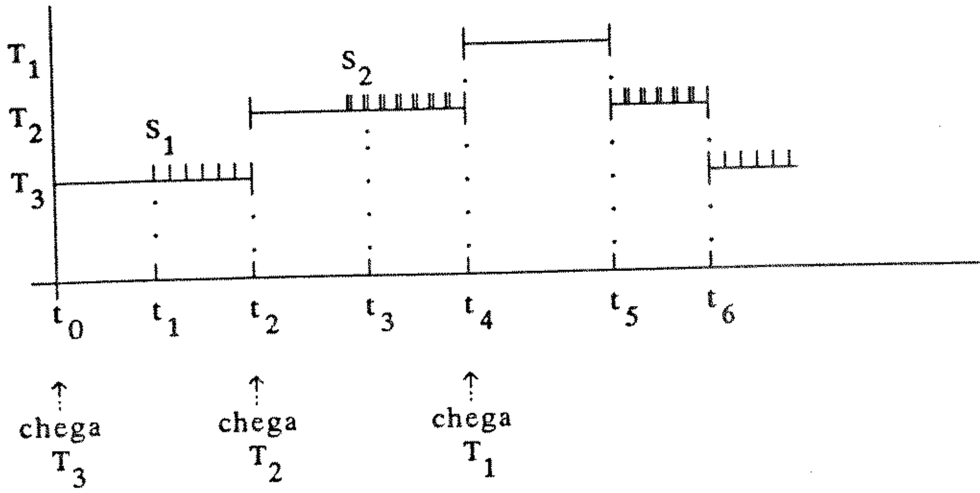
passo3 -- O processo espera na fila de prontos. Fim.

Note que o protocolo de herança de prioridade tem propriedade transitiva. Por exemplo (figura 3.2), suponha três processos T_1, T_2 e T_3; e prio(T_1) > prio(T_2) > prio(T_3). Considere um processo T_3 durante a execução de uma região crítica guardada pelo semáforo S_1. T_2 chega e faz a preempção sobre T_3. T_2 por sua vez sofre preempção pelo processo T_1 durante a execução de uma região crítica guardada por S_2. T_1 é bloqueado na tentativa de fechar o semáforo S_2, assim T_2 retoma a execução herdando a prioridade de T_1. Ao término da sua região crítica e na tentativa de entrar na região crítica guardada por S_1, T_2 é bloqueado e T_3 reassume a execução com a prioridade de execução de T_2 que por sua vez é a prioridade de T_1. Assim, diz-se que T_3 herda a prioridade de T_1 via T_2.

T_1= { p(s_2) v(s_2) }

T_2= { p(s_2) p(s_1) v(s_1) v(s_2) }

T_3= { p(s_1) v(s_1) }



→ execução da região crítica guardada pelo semáforo S_1

→ execução da região crítica guardada pelo semáforo S_2

→ execução do processo fora da região crítica

figura 3.2 - Propriedade de Transitividade do Protocolo de Herança de Prioridade

Podemos utilizar o **Exemplo I** da figura 3.1 e aplicar o algoritmo, teremos então a seguinte configuração :

(Exemplo II)

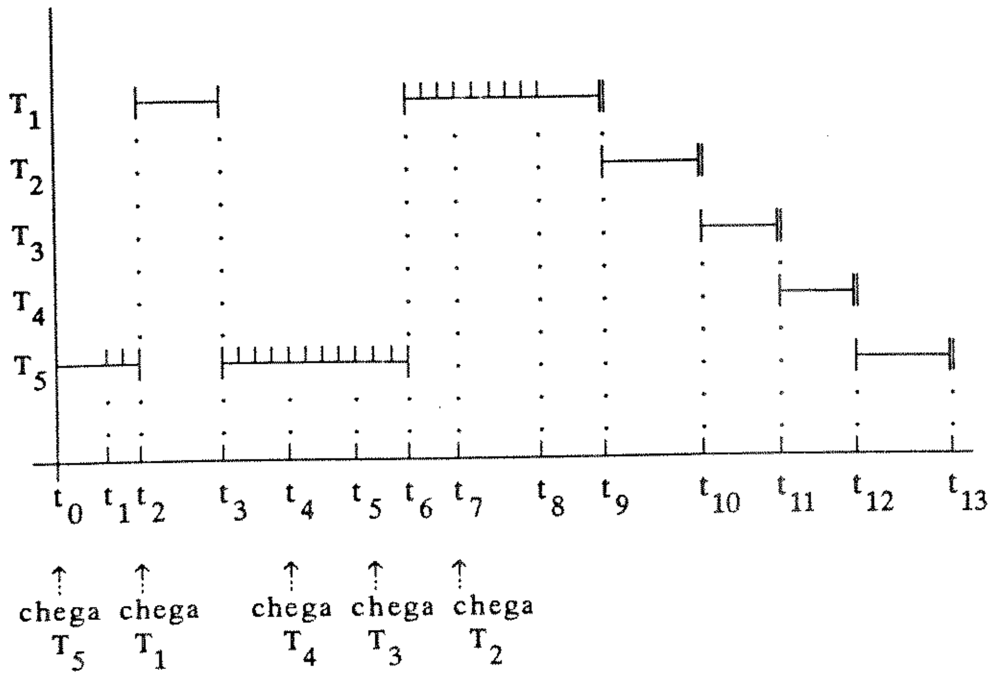


figura 3.3 - Funcionamento do Protocolo de Herança de Prioridade

t_0 : T_5 chega e executa

t_1 : T_5 requisita e consegue entrar na região crítica guardada pelo semáforo S, logo, T_5 executa a região crítica.

t_2 : T_1 chega e faz preempção em T_5 pois prio(T_1) é maior que prio(T_5). T_1 executa.

t_3 : T_1 solicita o acesso à região crítica mas s já foi fechado por T_5. Então T_1 é bloqueado por T_5 . T_5 herda a prioridade de T_1 : prio(T_5) = prio(T_1). T_5 continua a execução na região crítica.

t_4 : T_4 chega e fica na fila de prontos pois a prioridade herdada de T_5 é maior que a de T_4 . T_5 continua a execução.

t_5 : T_3 chega e fica na fila de prontos pois a prioridade herdada de T_5 é maior que a de T_3 . T_5 continua a execução.

t_6 : T_5 sai da região crítica, abre s e libera T_1. T_5 retoma sua prioridade original.

Como T_1 é o processo com a prioridade mais alta então T_1 executa. A tentativa de entrar na região crítica de T_1 é bem sucedido.

t_7 : T_2 chega e fica na fila de prontos.T_1 continua a execução.

t_8 : T_1 sai da região crítica e não existindo nenhum processo de prioridade mais alta para ser liberado, T_1 continua sua execução.

t_9 : T_1 finaliza a execução e T_2 toma a execução.

t_{10} : T_2 finaliza a execução e T_3 toma a execução.

t_{11} : T_3 finaliza a execução e T_4 toma a execução.

t_{12} : T_4 finaliza a execução e T_5 retoma a execução.

t_{13} : T_5 finaliza a execução.

É importante destacar que um processo de prioridade alta pode ser bloqueado por processo de baixa prioridade em duas situações :

I - *Bloqueio Direto*

Situações em que o processo mais prioritário tenta entrar na região crítica e o semáforo correspondente já foi fechado por outro processo de menor prioridade. Este é o preço a ser pago pela consistência dos dados compartilhados.

II - *Bloqueio de Herança*

Situações em que o processo de prioridade intermediária é bloqueado por um processo de prioridade mais baixa que herdou a prioridade de um processo de alta prioridade. Este é o preço pago para evitar que processos mais prioritários sejam bloqueados indiretamente por processos de prioridade intermediária. (Vide a figura 3.1)

3.2.1.3. Testes e Implementação

Usou-se o núcleo de tempo real DEADMOSI para verificar o problema citado acima e avaliar o protocolo de herança de prioridades.

Como os serviços em núcleos de tempo real tradicionais oferecem geralmente as primitivas simples de semáforos (wait e signal), as quais realizam apenas operações de abrir e fechar o semáforo e os bloqueios convencionais, o teste do protocolo só pode ser realizado se algumas modificações forem feitas. Mas, fazer novas adaptações num núcleo pronto é uma tarefa complicada principalmente para os usuários.

Assim, é proposto um algoritmo de controle das operações de wait e signal, o qual será incorporado aos processos do sistema, sem a necessidade de se modificar o núcleo. Este algoritmo engloba a criação das rotinas de controle que fazem tratamentos, como a herança e a retomada de prioridades, e os testes, tais como a verificação do estado (aberto ou fechado) do semáforo, necessários antes da execução dos comandos do núcleo.

Não se deve deixar de lembrar que o ambiente discutido no momento é o escalonador estático, o qual adota o esquema de prioridades pré-estabelecidas dos processos. As estruturas de dados e as rotinas que serão descritas abaixo devem ser codificadas e colocadas junto com o corpo dos processos.

3.2.1.3.1. Proposta da Implementação

São apresentadas duas estruturas principais de dados e duas rotinas.

>> ESTRUTURAS DE DADOS <<

(1) Uma fila de semáforos, cada um dos quais contendo seis campos de informações :

```
struct seminfo
{
 systag_t siden, piden;
 char estado;
 unsigned short priori;
 systag_t bloq[10];
 short total;
} semfila [ NUM_SEM ];
```

Os campos serão descritos a seguir :

siden ⟶ identificador do semáforo

piden ⟶ identificador do processo que fechou o semáforo siden

estado ⟶ indica se o semáforo está aberto ou fechado

priori ⟶ o campo guarda a prioridade original do processo piden

bloq[] ⟶ a fila armazena os processos bloqueados pelo processo piden

total ⟶ o número total de processos bloqueados

A fila **semfila[]** guarda, para cada semáforo, dados importantes para os testes de entrada e saída da região crítica.

(2) Um registro auxiliar que guarda resultados a serem transmitidos para o processo que deseja entrar ou sair da região crítica :

```
struct
{
 unsigned short  mudep;
 short param;
} tratinfo;
```

Esta estrutura de dados possui duas funções :

-**- No teste de entrada na região crítica, se a herança de prioridade deve ser feita, então este registro é usado para armazenar dados temporariamente.

tratinfo.mudep ⟶ guardar a prioridade a ser herdada

tratinfo.param ⟶ guardar o identificador do processo que vai herdar a prioridade guardada em mudep.

-**- No teste de saída da região crítica este registro terá outra utilidade :

tratinfo.mudep ⟶ guardar a prioridade original que deve ser retomada

tratinfo.param ⟶ guardar o número total de processos bloqueados que devem ser liberados.

>> ROTINAS DE TESTE <<

Duas rotinas **entraver** e **saiver** são criadas para uma série de testes antes de se efetuar os comandos WAIT e SIGNAL.

(1) Rotina entraver()

•• *Chamada* :

entraver(systag_t sema, unsigned short prioridade,systag_t identif);

•• *Parâmetros* :

sema ⟶ identificador do semáforo que guarda a região crítica

prioridade ⟶ prioridade do processo em **identif**

identif ⟶ identificador do processo que deseja entrar na região crítica guardada por sema

•• *Algoritmo da rotina* entraver() :

passo1) Encontrar a estrutura de dados que guarda informações referentes ao

semáforo **sema**

passo2) Verifica se sema está aberto ou fechado. Se **sema** estiver aberto então vai para passo3, caso contrário, para passo4

passo3) Marca "fechado" no campo **estado** e guarda os valores de **prioridade** e **identif** na estrutura de dados de **sema**. O que indica que o processo **identif** fechou o semáforo **sema** com prioridade **prioridade**. Retorna $\tilde{R}$ indicando não ser necessário a mudança de prioridade do processo. Fim.

passo4) Como **sema** já está fechado por outro processo então guarda **identif** na fila de bloqueados da estrutura de dados de **sema**. Encontra o processo, **piden**, que fechou **sema**. Armazena o processo **piden** que vai herdar a prioridade **prioridade** no registro **tratinfo**, nos campos **mudep** e **param**, respectivamente. Retorna R indicando a herança de prioridade. Fim.

(2) Rotina saiver()

•• *Chamada* :

saiver(systag_t sema);

•• *Parâmetros* :

sema ⟶ identificador do semáforo que guarda a região crítica da qual o processo está saindo.

•• *Algoritmo da rotina* saiver():

passo1) Encontrar a estrutura de dados que guarda as informações referentes ao semáforo sema

passo2) Marca "aberto" no campo **estado** de **sema**.

passo3) Se houver processos bloqueados na fila de bloqueio então vai para passo4, caso contrário vai para passo5

passo4) Armazenar no registro **tratinfo** o número de processos bloqueados e a prioridade original que o processo deve retomar nos campos **param** e **mudep**, respectivamente. Retorna **R** indicando a necessidade de retomada da prioridade original. Fim.

passo5) Armazena em **tratinfo.param** o valor 0 e retorna $\tilde{R}$ indicando que nenhum tratamento é necessário. Fim.

Deve ser observado que os atos de herança e de retomada de prioridade devem ser efetuados somente depois que as rotinas **entraver()** e **saiver()** retornaram o controle ao processo que as chamou, pois o comando que as realizam provoca uma reordenação dos processos na fila de prontos, o que pode resultar em uma situação indesejável na qual os processos que solicitou a entrada ou saída da região crítica, perde indevidamente o controle de execução.

Dentro de cada processo que compartilha uma área comum são necessários algumas adaptações, como as mostradas a seguir :

(I) - Antes de entrar na região crítica

O objetivo desta proposta é simular, a nível do usuário, um pseudo-núcleo que aplica o protocolo de herança de prioridade, quando um processo quer entrar na região crítica ou sair dela.

Para implementar este protocolo em um núcleo real, as verificações e os tratamentos necessários devem ser embutidos nos comandos **WAIT** e **SIGNAL**, o que implica que durante suas execuções o núcleo não permitirá qualquer preempção de outro processo. Porém, não é aconselhável o uso da técnica de desabilitação de interrupção para este fim, pois pode violar o sentido de um sistema de tempo real afetando a concorrência de processos de outro módulo. Portanto, criamos um nível especial de prioridade mais alta dos processos envolventes para proteger as verificações dos semáforos pelos processos em execução.

Desse modo, um processo, antes de tentar entrar numa região crítica, altera a sua prioridade atual **Prio_Atual** para **PRIO_ALTA**, a mais alta dos processos, para somente então analisar a situação do semáforo e, caso este esteja aberto, fechá-lo entrando assim na região crítica, após retornar à sua prioridade **Prio_Atual**.Se o semáforo estiver fechado (existe outro processo dentro da região crítica) então é realizada a herança de prioridade do processo que está na região crítica. Logo, o processo retorna à sua prioridade **Prio_Atual** (com a qual o processo solicitou a entrada da região crítica) e fica bloqueado à espera da liberação da região crítica.

Suponha um processo T_i que tem prioridade $prio_i$ no momento de solicitar a entrada na região crítica guardada pelo semáforo s_i

```
        •
        •
        •

    varlocal = True
    Enquanto varlocal = True faça
  ┌──── Mudar a prioridade para PRIO_ALTA
  │         └→ oschtyp(Ti, PRIO_ALTA, ---)
  │     Teste para entrar na região crítica
  │         └→ valor = entraver(si, prioi, Ti)
  │     Se valor indica que o semáforo si está aberto
  │     Então
Parte           varlocal = False
  │             Fechar o semáforo
Protegida          └→ oswais(si,---)   : comando WAIT
  │     Senão /* o semáforo si já foi fechado */
  │             Efetuar a herança de prioridade do processo
  │             que fechou o semáforo si
  │                └→ oschtyp(tratinfo.param, tratinfo.mudep,---)
  │     Fim-Se
  └──── Retornar à sua prioridade prioi
           └→ oschtyp(Ti, prioi, ---)
        Se varlocal = True    /* Ti vai ser bloqueado */
        Então
                Ficar bloqueado
                   └→ oswais(si,---)   : comando WAIT
        Fim-Se
    Fim-Enquanto
        •
        •
        •
```

figura 3.4 - Tratamentos de um Processo durante a Solicitação de Entrada de R.C. - Protocolo de Herança de Prioridade

|| *Observação* ||

A variável "varlocal" deve ser local para cada processo de modo que não seja alterada pelos outros processos. Na verdade, ela é usada para o controle do looping "*Enquanto*", que por sua vez possui uma sequência de comandos que representa a tentativa de entrar na região crítica guardada por s_i. Essa tentativa é repetida pelo processo até que este encontre o semáforo s_i aberto.

A maior parte do teste precisa ser protegida contra qualquer preempção de outros processos que utilizam o mesmo semáforo. Assim, um nível de prioridade mais alta, como citado anteriormente, deve ser criado para este fim e somente usado pelo processo na hora de tentar entrar na região crítica.

Observe que a mudança, o retorno e a herança de prioridade são realizados através do comando do núcleo DEADMOSI "Change Type".

Quando o processo verificou, dentro de "Parte Protegida" na figura 3.4, que o semáforo s_i está aberto, o comando WAIT é usado dentro de "Parte Protegida", sem sofrer preempção de forma alguma, para fechar o semáforo e logo em seguida, o processo sai de "Parte Protegida" e do looping "*Enquanto*", entrando assim na região crítica. Se o semáforo s_i já foi fechado (situação verificada pelo processo dentro de "Parte Protegida"), então o processo que fechou s_i herda a prioridade do processo T_i, através do comando "Change Type". Neste caso, o comando WAIT é usado pelo processo T_i para se bloquear fora de "Parte Protegida". A razão desta providência é evitar a situação complicada, na qual o processo fica bloqueado com a prioridade especial que não é a sua prioridade real.

(II) - Antes de sair da região crítica

O teste na saída da região crítica, a nível de usuário, de um processo é menos complicado que o da entrada pelo fato de que um único processo pode sair de uma região crítica e abrir o semáforo correspondente. Contudo, a proteção de certos tratamentos também é necessária. Usa-se então a mesma técnica de proteção da entrada na região crítica (criar um nível especial de prioridade alta) contra as preempções indesejáveis.

Suponha o processo T_i que tem prioridade $prio_i$ no momento de solicitar a saída da região crítica guardada pelo semáforo s_i

```
        •
        •
        •

         Mudar a prioridade para PRIO_ALTA
           └→ oschtyp(Ti, PRIO_ALTA)
          Teste para sair da região crítica
            └→ valor = saiver(Si)
          Liberar todos os processos bloqueados nete semáforo
Parte       └→ osmsigs(si, tratainfo.param + 1, ---)
          Se valor indica que precisa retomar sua prioridade
          original
Protegida Então
               Passar as informações da estrutura global para
               os variáveis locais
                  └→ ret_prio = tratinfo.mudep
               varlocal = True
          Fim-Se
         Retornar a prioridade prioi
         Se varlocal = True /* tinha prioridade herdada  */
         Então
               Retomar a sua prioridade original do processo
                 └→  oschtyp(Ti, ret_prio)
         Fim-Se
               •
               •
               •
```

figura 3.5 - Tratamentos de um Processo durante a Solicitação de Saída de R.C. - Protocolo de Herança de Prioridade

|| *Observação* ||

A variável **varlocal** é local para cada processo e indica a necessidade ou não da retomada de prioridade do processo após a saída da região crítica.

A "Parte Protegida" consiste no teste da rotina **saiver()**, na liberação de todos os processos bloqueados pelo semáforo s_i e na passagem de prioridade original que o processo T_i deve retomar, da estrutura global **tratainfo.mudep** para a variável local **ret_prio**. O último é importante para manter a consistência do valor de prioridade original depois de sair da "Parte Protegida".

Apesar do núcleo DEADMOSI ordenar os processos bloqueados num semáforo por prioridade, a organização pela sequência de bloqueio dos processos é a situação mais comumente encontrada num núcleo convencional. Esta política de organização possibilita a liberação de um processo menos prioritário, se este for o próximo processo na fila de bloqueados. Para evitar este problema, todos os processos da fila de bloqueio são liberados, proporcionando uma concorrência por prioridade para entrar na região crítica.

Como foi descrito no algoritmo **entraver()**, a estrutura de dados "**tratinfo.param**" contém o número de processos bloqueados que, neste caso, devem ser liberados. Usa-se o comando "Change Type" para a mudança da prioridade para o nível especial, **PRIO_ALTA**, antes de executar "Parte Protegida" e na retomada da prioridade original do processo T_i, caso este tenha herdado a prioridade de algum outro processo. A execução da liberação dos processos, antes da retomada da prioridade, deve ser respeitada pois o comando "Change Type" provoca um reescalonamento dos processos no sistema, o que pode levar à perda do controle da execução na CPU.

Na sequência são apresentados alguns lemas e teoremas que resultam das análises de propriedades deste protocolo.

3.2.1.4. Lemas e Teoremas

Lema 1

Um processo T_B de prioridade baixa pode bloquear o processo T_A, de prioridade alta, no máximo pelo equivalente à duração de uma execução da região crítica de T_B , sem considerar o número de semáforos que T_B e T_A compartilham.

Prova

Para que T_A seja bloqueado por T_B, este precisaria ter entrado na região crítica antes de T_A e ainda permanecer nela quando T_A chegar. Assim que T_B sai da região crítica o processo T_A ou outros processos de prioridade maior que a de T_B toma o controle de execução imediatamente. E a partir desse momento, como T_B não está mais na região crítica e T_A ficou pronto no sistema, o processo não mais terá chance de bloquear T_A

Observação

A garantia oferecida pelo lema 1 é válida para uma única ocorrência do processo T_A, ou seja, durante a execução de cada instância.

Teorema 1

Neste protocolo, se existem n processos de prioridade mais baixa que a do processo T, então ele pode ser bloqueado no máximo pela duração de n regiões críticas.

Prova

Segundo o lema 1 cada um dos n processos pode bloquear no máximo o equivalente à duração de uma região crítica, logo, o teorema é válido.

Note que na figura 3.2, o processo T_1 é bloqueado, no pior caso, pelo equivalente de duas regiões críticas de T_2 e T_3 se T_2 chega logo depois que T_3 entrar na região crítica no instante t_1 e T_1 chega pouco após o instante t_3.

Lema 2

Seja S o semáforo que pode causar bloqueios diretos ou bloqueios de herança ao processo T. Então S pode ser usado para bloquear T no máximo o equivalente à uma região crítica de um processo de prioridade mais baixa.

Prova

Suponha que S pode ser usado para bloquear um processo T pela duração de n regiões críticas. Neste caso, pelo lema 1, deve existir n processos de prioridade mais baixa que a de T que usam o mesmo semáforo S para bloquear T. Como já se sabe, uma das condições para que um processo mais prioritário seja bloqueado por outro processo menos prioritário é que este esteja na região crítica. Portanto, isto implica a coexistência de n processos na mesma região crítica o que é contraditório para a definição de uma região crítica.

Teorema 2

Na aplicação do protocolo de herança de prioridade, se existem m semáforos que podem ser usados para bloquear o processo T então T pode ser bloqueado no máximo pelo equivalente à duração de m regiões críticas.

Prova

Segundo o lema 2, cada um dos m semáforos só pode ser usado para bloquear T pelo equivalente à duração de uma região crítica, logo, o teorema se justifica.

3.2.2. Problemas de "Deadlock" e de Cadeia de Bloqueios

O protocolo básico de herança de prioridade não prevê os problemas de "deadlock" ou o bloqueio mútuo, e a formação de cadeia de bloqueios.

A situação de "deadlock" é um problema importante nos sistemas de processos concorrentes onde dois processos podem gerar bloqueios mútuos, o que implica numa "trava mortal" ao sistema. O exemplo abaixo (figura 3.6)fornecerá uma visão mais clara do problema de "deadlock" quando o protocolo de herança de prioridade é aplicado.

(Exemplo III)

Sejam T_1 e T_2 dois processos que compartilham duas regiões críticas diferentes guardadas por S_1 e S_2. As regiões críticas são aninhadas como mostradas a seguir :

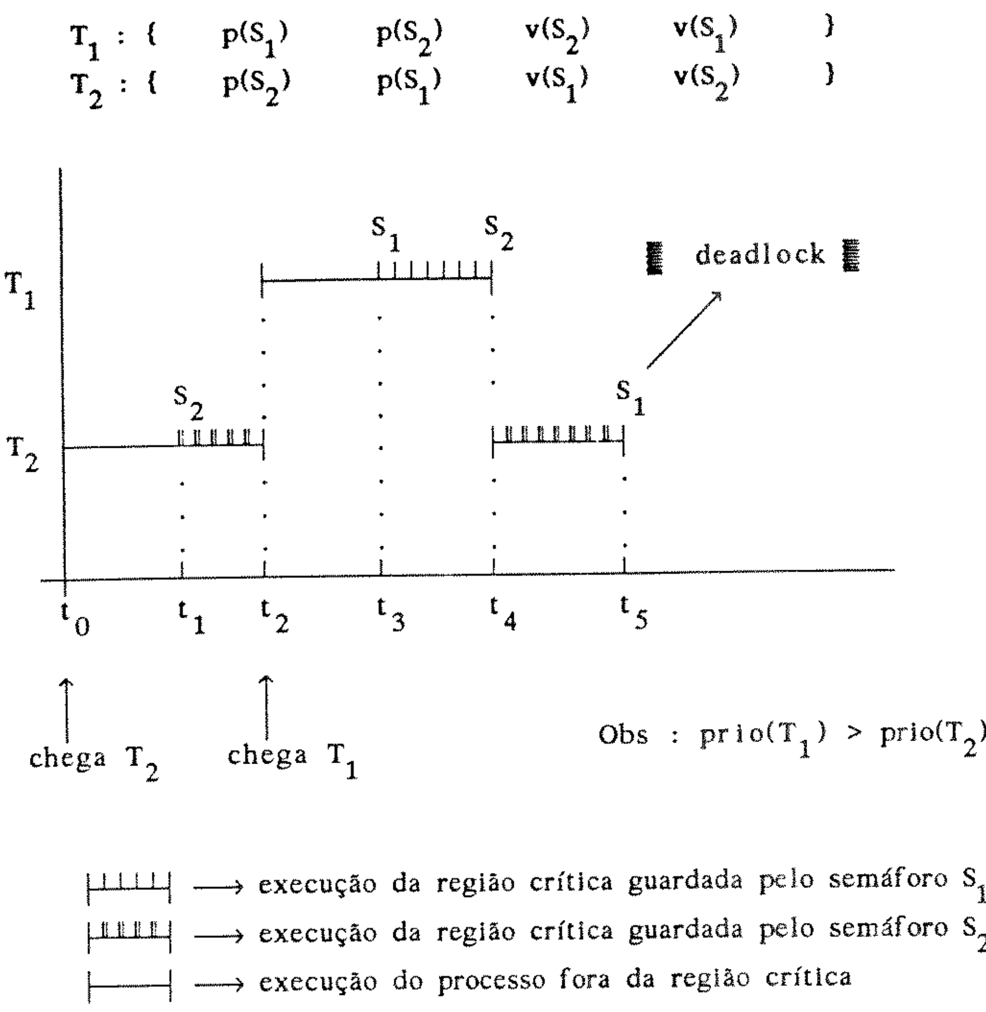


figura 3.6 - Problema de "Deadlock" do Exemplo III

t_0 : T_2 chega e executa

t_1 : T_2 entra na região crítica e fecha o semáforo S_2

t_2 : T_1 chega e faz preempção em T_2 . T_1 executa pois prio(T_1) é maior que prio(T_2)

t_3 : T_1 entra na região crítica guardada por S_1 e fecha o semáforo

t_4 : T_1 queria acessar a região crítica aninhada guardada por S_2 mas S_2 já foi fechado por T_2 então T_1 é bloqueado por T_2. T_2 retorna à execução herdando a prioridade

de T_1

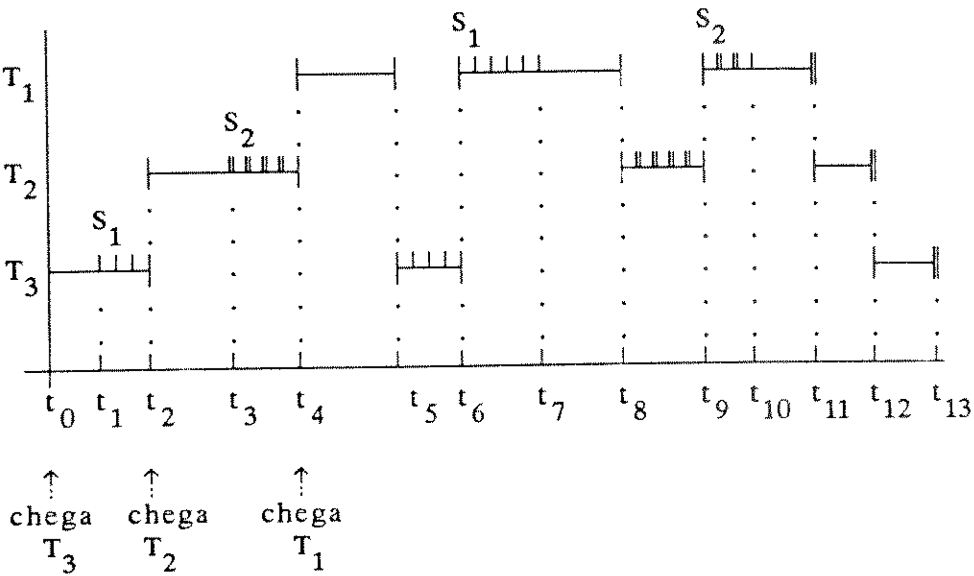
t_5 : T_2 queria acessar sua região crítica aninhada e guardada por S_1. Porém, S_1 já foi fechado pelo processo T_1 então T_2 agora é bloqueado por T_1

Assim, na figura 3.6, o bloqueio mútuo ou deadlock entre os processos T_1 e T_2 é formado.

O segundo problema é a formação de cadeias de bloqueios. Também a situação será ilustrada através do exemplo a seguir (figura 3.7) :

(Exemplo IV)

T_1 : { $p(S_1)$ $v(S_1)$ $p(S_2)$ $v(S_2)$ }
T_2 : { $p(S_2)$ $v(S_2)$ }
T_3 : { $p(S_1)$ $v(S_1)$ }



⟶ execução da região crítica guardada pelo semáforo S_1

⟶ execução da região crítica guardada pelo semáforo S_2

⟶ execução do processo fora da região crítica

figura 3.7 - Problema de Cadeia de Bloqueios do Exemplo IV

t_0 : T_3 chega e executa

t_1 : T_3 é bem sucedido na requisição de entrar na região crítica guardada pelo semáforo S_1, logo, T_3 executa a região crítica.

t_2 : T_2 chega e faz preempção em T_3 pois prio(T_2) é maior que prio(T_3). T_2 executa.

t_3 : T_2 entra na região crítica guardada por S_2 fechando-o.

t_4 : T_1 chega e faz preempção em T_2 pois prio(T_1) > prio(T_2) T_1 executa.

t_5 : T_1 queria entrar na região crítica guardada por S_1 mas como S_1 já foi fechado por T_3 então T_1 é bloqueado por T_3 . T_3 retorna à execução herdando a prioridade de T_1.

t_6 : T_3 sai da região crítica, libera S_1 e T_1 ; T_3 retoma sua prioridade original. Como T_1 é o processo com prioridade mais alta, ele executa. A tentativa de entrar na região crítica de S_1 é agora bem sucedido.

t_7 : T_1 sai da região crítica guardada por S_1 e continua a execução.

t_8 : T_1 queria entrar na região crítica guardada por S_2 mas S_2 já foi fechado por T_2 então T_2 toma controle de execução e herda a prioridade de T_1; T_1 é bloqueado outra vez.

t_9 : T_2 sai da região crítica e libera T_1; T_1 recupera a execução e entra na região crítica guardada por S_2

t_{10} : T_1 sai da região crítica e continua sua execução.

t_{11} : T_1 finaliza a execução e T_2 toma a execução.

t_{12} : T_2 finaliza a execução e T_3 retoma a execução.

t_{13} : T_3 finaliza a execução.

Note que nos instantes t_5 e t_8 o processo T_1 foi bloqueado duas vezes por processos diferentes (T_3 e T_2 respectivamente). Portanto, a existência do problema de cadeia de bloqueios é justificada.

Diante desses dois problemas o chamado protocolo de prioridade topo é empregado para resolvê-los e tentar um melhor desempenho no sistema com escalonador estático. O novo protocolo não somente minimiza o tempo de bloqueio do processo T por, no máximo, a duração de uma região crítica, mas, também evita a formação de deadlock.

3.2.2.1. Definição do Protocolo de Prioridade Topo

Define-se prioridade topo de um semáforo S como sendo a prioridade mais alta entre as de todos os processos que compartilham uma região crítica guardada por S.

O objetivo principal deste protocolo é assegurar que um processo T_i somente possa entrar numa região crítica se a sua prioridade for estritamente maior que a maior prioridade topo dos semáforos já fechados no sistema; em outras palavras, T_i só executa uma região crítica com a certeza de não atrasar outros processos de maior prioridade que compartilham uma região crítica cujo semáforo já tenha sido fechado.

3.2.2.2. Algoritmo do Protocolo

Na verdade, é somente em alguns pontos da transação, tais como na tentativa de se entrar na região crítica, na saída da região crítica e na chegada de um processo, que os tratamentos devem ser feitos. A seguir será descrito o procedimento, destacando-se estas situações.

■■ Quando um processo T quer acessar a região crítica guardada pelo semáforo S

S^* : o semáforo dentre os já fechados que possui a maior prioridade topo exceto os semáforos fechados pelo prório T.

T^* : o processo que fechou o semáforo S^*

passo1 -- Encontrar o semáforo S^* e verificar se a prioridade do processo **T** é maior que a prioridade topo de S^*. Caso afirmativo, vai para passo2, senão vai para passo3.

passo2 -- O processo **T** entra na região crítica. **T** fecha o semáforo e executa a região crítica; fim.

passo3 -- O processo **T** não consegue entrar na região crítica e fica guardado na fila de bloqueios do processo T^* o qual tinha fechado S^* . T^* herda a prioridade de T. Fim.

■■ **Quando o processo T sai da região crítica guardada por S :**

passo1 -- Abre o semáforo associado S que foi fechado por T

passo2 -- Libera todos os processos bloqueados por **T**, se tiver algum.

passo3 -- Se **T** tinha herdado a prioridade de algum processo de prioridade mais alta então retoma sua prioridade original.

■■ **Quando o processo T chega e o processo T' está sendo executado :**

passo1 -- Verificar se a prioridade de **T** é mais alta que a prioridade, herdada ou original, com a qual **T'** está executando. Caso afirmativo, vai para passo2, senão vai para passo3.

passo2 -- O processo **T** faz preempção em **T'** . Fim.

passo3 -- O processo **T** espera na fila de prontos. Fim.

(Exemplo V)

A partir do **(Exemplo III)** deste capítulo (figura 3.6) aplica-se o protocolo de prioridade topo a fim de se eliminar o problema de deadlock. A sequência de execução de cada processo é :

T_1 : { $p(S_1)$ $p(S_2)$ $v(S_2)$ $v(S_1)$ }

T_2 : { $p(S_2)$ $p(S_1)$ $v(S_1)$ $v(S_2)$ }

Tem-se "$pceil(S_i)$" como a prioridade topo do semáforo S_i e, "$prio_i$" e "$prio(T_i)$" como a prioridade original e de execução do processo T_i , respectivamente, sendo $prio_1 > prio_2$. Inicialmente, tem-se então :

$pceil(S_1) = prio_1$ $prio(T_1) = prio_1$

$pceil(S_2) = prio_1$ e $prio(T_2) = prio_2$

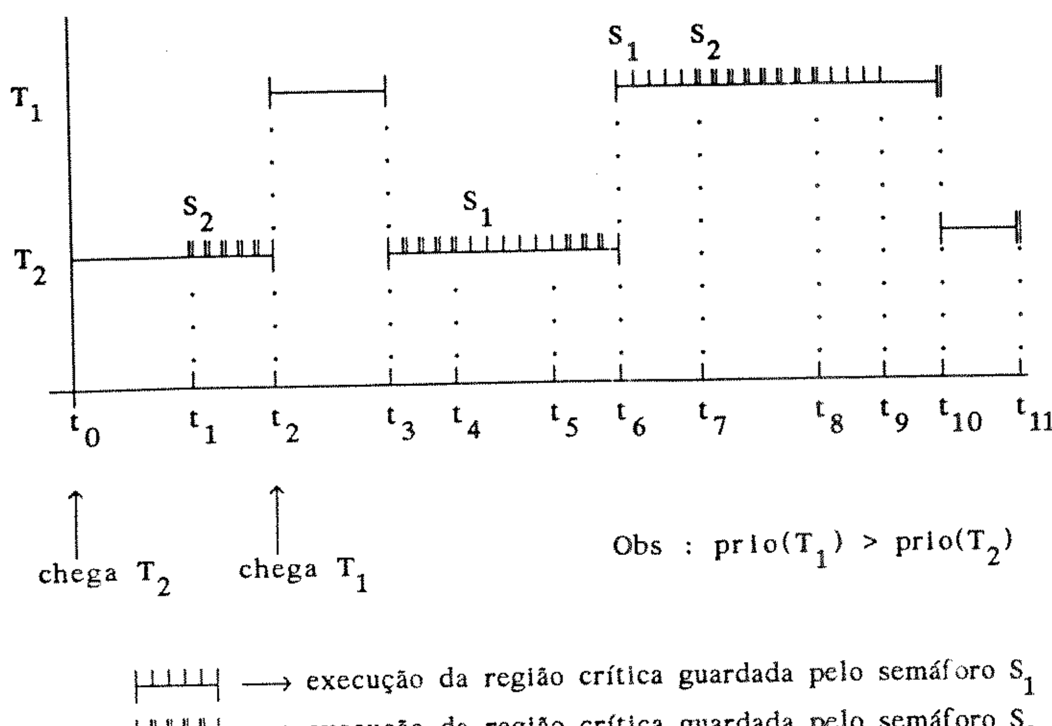


figura 3.8 - Problema de "Deadlock" Resolivido usando o Protocolo de Prioridade Topo

t_0 : T_2 chega e executa

t_1 : T_2 requisita e consegue entrar na região crítica guardada pelo semáforo S_2; logo, T_2 executa a região crítica.

t_2 : T_1 chega e faz preempção em T_2 pois prio(T_1) é maior que prio(T_2). T_1 executa.

t_3 : T_1 deseja entrar na região crítica guardada por S_1, mas existe um semáforo fechado, S_2 , por T_2 , e prio(T_1) não é maior que pceil(S_2); logo, T_1 é bloqueado e T_2 retoma a execução. T_2 herda a prioridade de T_1 : prio(T_2) = $prio_1$

t_4 : T_2 entra na região crítica guardada por S_1, pois não existe nenhum semáforo fechado além de S_2, o qual foi fechado pelo próprio T_2 , $S^* = \phi$.

t_5 : T_2 sai da região crítica, libera S_1 e continua a execução na região crítica guardada por S_2 .

t_6 : T_2 sai da região crítica, libera S_2 e recupera sua prioridade original : prio(T_2) = $prio_2$. T_1 é liberado e como prio(T_1) > prio(T_2), T_1 toma a execução da região crítica guardada por S_1.

t_7 : T_1 entra na região crítica guardada por S_2, pois não existe semáforo fechado além dos fechados por T_1

t_8 : T_1 sai da região crítica guardada por S_2 e continua a execução na região crítica de S_1.

t_9 : T_1 sai da região crítica abrindo o semáforo S_1. Continua executando.

t_{10} : T_1 finaliza a execução . T_2 retoma a execução.

t_{11} : T_2 finaliza a execução.

Assim, o problema de deadlock é contornado. Vamos aplicar o protocolo de prioridade topo no **(Exemplo IV)** para tentar resolver o problema de bloqueio múltiplo.

(Exemplo VI)

Utiliza-se o **(Exemplo IV)** deste capítulo (figura 3.7), e a sequência de execução de cada processo é :

T_1 : { $p(S_1)$ $v(S_1)$ $p(S_2)$ $v(S_2)$ }
T_2 : { $p(S_2)$ $v(S_2)$ }
T_3 : { $p(S_1)$ $v(S_1)$ }

acrescentando-se os dados sobre as prioridades topo dos semáforos :

$pceil(S_1) = prio_1$ $prio(T_1) = prio_1$
$pceil(S_2) = prio_1$ $prio(T_2) = prio_2$
$prio(T_3) = prio_3$

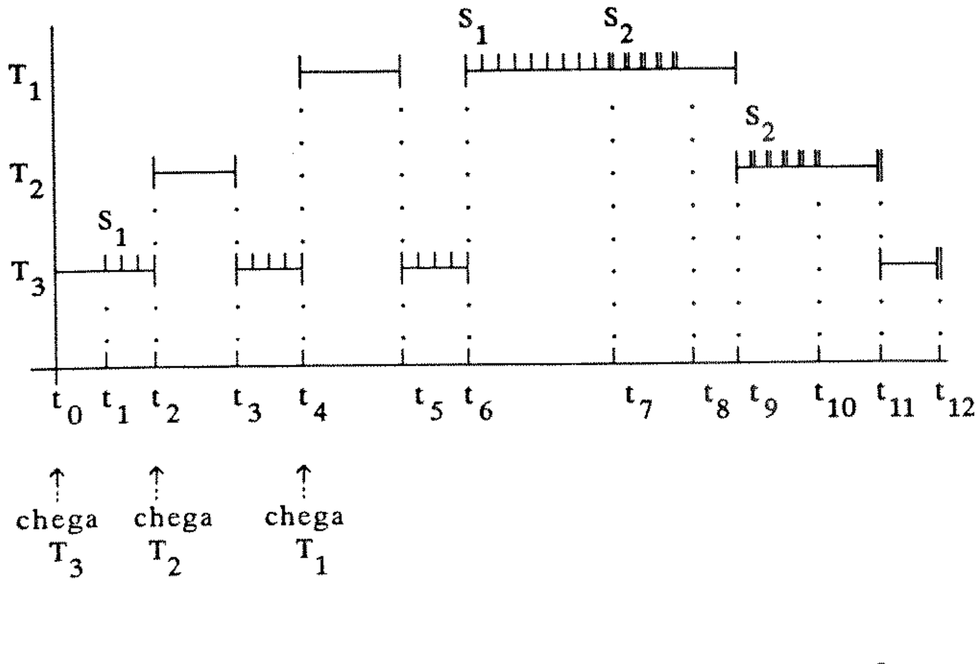


figura 3.9 - Problema de Cadeia de Bloqueios Resolvido usando Protocolo de Prioridade Topo

t_0 : T_3 chega e executa

t_1 : T_3 é bem sucedido na requisição de entrar na região crítica guardada pelo semáforo S_1; logo, T_3 executa a região crítica.

t_2 : T_2 chega e faz preempção em T_3 pois prio(T_2) é maior que prio(T_3); T_2 executa.

t_3 : T_2 não consegue entrar na região crítica pois sua prioridade $prio_2$ não é estritamente maior que a prioridade topo do semáforo fechado S_1 ($prio_2$ < pceil(S_1)=$prio_1$). T_2 é bloqueada e T_3 herda a prioridade de T_2 : prio(T_3) = $prio_2$; T_3 retoma sua execução da região crítica.

t_4 : T_1 chega e faz preempção em T_3 pois a prioridade de T_1 é maior que a prioridade herdada do processo T_3 :$prio_1$ > prio(T_3)=$prio_2$; T_1 executa.

t_5 : T_1 não consegue entrar na região crítica pois a prioridade de T_1 não é estritamente maior que a prioridade topo do semáforo S_1 que foi fechado por T_3. T_1 então é bloqueada por T_3 e T_3 herda a prioridade de T_1 : prio(T_3) = $prio_1$. T_3 retoma a execução da região crítica.

t_6 : T_3 sai da região crítica, abre S_1. Os processos T_1 e T_2 são colocados na fila de prontos;T_3 recupera sua prioridade original. Como T_1 é o processo com prioridade mais alta, ele executa. A tentativa de entrar na região crítica de S_1 é agora bem sucedida.

t_7 : T_1 sai da região crítica guardada por S_1 e consegue entrar na região crítica guardada por S_2 pois $S^* = \phi$

t_8 : T_1 sai da região crítica guardada por S_2 e continua a sua execução

t_9 : T_1 finaliza sua execução, T_2 retoma o controle de CPU e consegue entrar na região crítica guardada por S_2 pois $S^* = \phi$

t_{10} : T_2 sai da região crítica e continua sua execução.

t_{11} : T_2 finaliza a execução e T_3 retoma a execução.

t_{12} : T_3 finaliza a execução.

Deve ser observado que o processo T_1 só foi bloqueado uma única vez no instante t_5 , em contraste com o **(Exemplo IV)**, em que o processo T_1 foi bloqueado duas vezes por dois processos diferentes. Assim, o problema de bloqueio múltiplo é resolvido.

3.2.2.3. Testes e Implementação

O esquema da implementação deste protocolo, no núcleo DEADMOSI, é análogo ao do protocolo de herança de prioridades descrito no ítem anterior, exceto algumas adaptações adicionais tais como uma nova fila de semáforos fechados e um teste de prioridade topo com a prioridade do processo que requer a entrada na região crítica.

3.2.2.3.1. Proposta da Implementação

Primeiramente serão apresentados três estruturas de dados principais e quatro rotinas.

>> ESTRUTURAS DE DADOS <<

(1) Uma fila de semáforos do sistema, contendo a identificação do semáforo e sua prioridade topo.

```
struct sinfo
{
  systag_t iden;
  unsigned short  ptopo;
} semaforo [N_SEM];
```

Os campos de informação são descritos como :

iden ⟶ identificação do semáforo

ptopo ⟶ prioridade topo do semáforo

(2) Uma fila de semáforos fechados cada um dos quais contendo cinco campos de informação :

```
struct sf
{
 struct sinfo *pontsem;
 systag_t pid;
 unsigned short  porigem;
 short tnum;
 struct sf  *prox;
} semlock [ MAX_LOCK ];
```

Os campos são descritos como :

pontsem ⟶ ponteiro que aponta para o registro, da fila **semaforo[]**, que possui as informações do semáforo referido, **semaforo.iden**

pid ⟶ processo que fechou o semáforo **semaforo.iden**

porigem ⟶ prioridade original do processo **pid**

tnum ⟶ número total de processos bloqueados por **semaforo.iden**

prox ⟶ aponta para o próximo registro de semáforo fechado

A fila **semaforo[]** é inicializada com as identificações dos semáforos existentes no sistema e suas prioridades topo. Quando um processo T fechar um semáforo S então um registro da fila **semlock[]** é alocado para este semáforo e o campo do ponteiro **pontsem** é preenchido com o endereço do registro que contém as informações do semáforo S. A figura 3.10 mostra essa configuração das estruturas de dados.

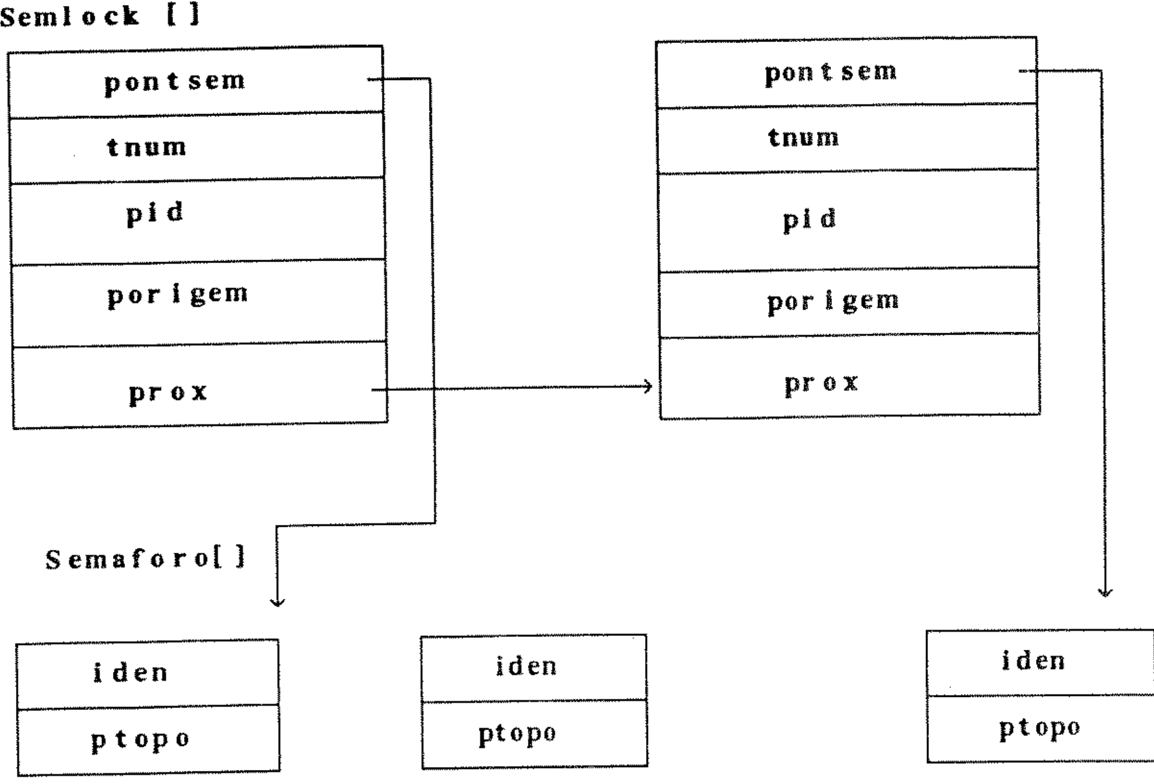


figura 3.10 - Ligação entre as duas Filas Semlock[] e Semaforo[]

(3) Uma estrutura de dados, guardando as informações necessárias quando as rotinas de teste **entraver** e **saiver** retornam ao processo que solicita a entrada e saída da região crítica.

```
struct
{
 unsigned short  mudep;
 short  param;
 systag_t  proc;
} guarda;
```

Os campos deste registro serão descritos a seguir :

mudep ⟶ guardar a prioridade que será herdada ou retomada pelo processo

param ⟶ guardar o número de processos bloqueados que serão liberados ou a identificação do semáforo que bloqueia o processo proc

proc ⟶ a identificação do processo que será bloqueado

>> ROTINAS DE TESTE DE SEMÁFORO <<

(1) <u>Rotina</u> <u>maxprio()</u>

•• *Chamada da rotina*

***maxprio(systag_t proid);**

•• *Parâmetros*

proid ⟶ identificação do processo que solicitou a entrada na região crítica

•• *Objetivo principal*

Caso existam semáforos fechados, que não tenham sido fechados pelo processo que deseja entrar na região crítica, seleciona aquele de maior prioridade topo.

•• *Valor de retorno*

NULL ⟶ não encontrou o semáforo referido

não NULL ⟶ retorna o endereço do registro que contém informações sobre o semáforo referido

(2) <u>Rotina</u> <u>alocsem()</u>

•• *Chamada*

***alocsem(systag_t sid)**

•• *Parâmetros*

sid ⟶ identificação do semáforo que guarda a região crítica na qual o processo solicita entrada

•• *Objetivo principal*

Alocar um registro de semáforo para inserção na fila de semáforos fechados correspondente, de acordo com a prioridade topo do semáforo **sid**

•• *Valor de retorno*

Retornar o ponteiro que aponta para o registro do semáforo que o processo acabou de fechar

(3) <u>Rotina</u> <u>entraver()</u>

•• *Chamada*

entraver(systag_t sid, unsigned short prioridade,

systag_t proid);

•• *Parâmetros*

sid ⟶ identificação do semáforo que o processo **proid** quer fechar

prioridade ⟶ prioridade do processo **proid**

proid ⟶ processo que deseja entrar na região crítica

•• *Algoritmo*

passo1) Achar entre os semáforos fechados por outros processos que não o **proid**, o de maior prioridade topo **s = maxprio(proid)**

passo2) Se o valor de retorno **s** é NULL ou a prioridade do processo **proid** é maior que a prioridade topo de **s**, então vai para passo4 senão segue para o próximo passo.

passo3) Processo **proid** não pode entrar na região crítica; logo, deve-se atualizar o número de bloqueios do semáforo s.Guardar a prioridade do processo **proid** e o semáforo **s** nos campos **mudep** e **param** e retorna **R**, indicando o bloqueio de **proid** e a herança de prioridade do processo que fechou **s**.

passo4) É permitido ao processo entrar na região crítica. Alocar então o registro de semáforo fechado e inseri-lo na fila.

v = alocsem(sid)

Atribuir as informações necessárias ao registro alocado e retornar $\tilde{R}$ indicando o sucesso de entrar na região crítica do processo **proid**

(4) <u>Rotina</u> <u>saiver()</u>

•• *Chamada*

saiver(systag_t sid, unsigned short prioridade)

•• *Parâmetros*

sid ⟶ identificação do semáforo da região crítica que deve ser aberto antes do processo sair

prioridade ⟶ prioridade de execução com a qual o processo está saindo da região crítica

•• *Algoritmo*

passo1) Achar na fila de semáforos fechados o registro que contém as informações do semáforo **sid**

passo2) Armazenar a prioridade original do processo, que queria sair da região crítica, no campo **mudep**, e o número de processos bloqueados que serão liberados, no campo **param** da estrutura de dados **guarda**.

passo3) Zerar os campos do registro de semáforo e devolvê-lo à fila de semáforos disponíveis

passo4) Se a prioridade de execução do processo é maior que a original então

retorna **R**, indicando a necessidade da retomada da prioridade original do processo, senão retorna $\tilde{R}$ indicando a situação contrária.

(I) - Antes de entrar na região crítica

A idéia de simular, a nível de usuário, um pseudo-núcleo que aplica o protocolo de prioridade topo é similar à do protocolo de herança de prioridade. Um nível de prioridade mais alta deve também ser criado para que,quando um processo solicitar a entrada na região crítica, a verificação e os tratamentos adequados possam ser feitos sem sofrer preempção de outros processos em hipótese alguma.

Sejam, T_i o processo que possui prioridade $prio_i$ no momento de solicitar a entrada na região crítica guardada pelo semáforo s_i, e sem', o semáforo de maior prioridade topo entre os fechados mas não fechado por T_i (figura 3.11).

```
        •
        •
        •

   varlocal = True
   Enquanto varlocal = True faça
   ┌──── Mudar a prioridade para  PRIO_ALTA
   │          └→ oschtyp(Ti, PRIO_ALTA, ---)
   │       Teste para entrar na região crítica
   │          └→ valor = entraver(si, prioi, Ti)
   │       Se valor indica que o semáforo si estar aberto
   │       Então
 Parte            varlocal = False
   │              Fechar o semáforo si
Protegida            └→ oswais(si,---)  : comando WAIT
   │       Senão /* Ti não é permitido a entrar na R.C. */
   │              Efetuar a herança de prioridade do processo
   │              que fechou o semáforo sem'
   │                 └→ oschtyp(tratinfo.param, tratinfo.mudep,---)
   │       Fim-Se
   └──── Retornar à sua prioridade prioi
             └→ oschtyp(Ti, prioi, ---)
           Se varlocal = True    /* Ti vai ser bloqueado */
           Então
                  Ficar bloqueado
                     └→ oswais(sem', ---)   : Comando WAIT
           Fim-Se
   Fim-Enquanto
        •
        •
        •
```

figura 3.11 - Tratamentos de um Processo durante a Solicitação de Entrada de R.C. - Protocolo de Prioridade Topo

|| *Observação* ||

O variável "**varlocal**" deve também ser local para cada processo. (vide o ítem correspondente do protocolo de herança de prioridade). O looping "*Enquanto*" é usado para repetir a tentativa de entrar na região crítica, até que o processo T_i finalmente obtenha essa permissão. E a função de "*Parte Protegida*" é idêndica à do item referido.

Apesar da grande semelhança deste teste com o do protocolo de herança de prioridades, existe uma diferença que deve ser destacada : no protocolo de herança de prioridades um processo relaciona-se com outros processos, através do compartilhamento de uma mesma região crítica e as consultas aos semáforos tem este fato por base. Já no protocolo de prioridade topo, as consultas levam em conta todas as regiões críticas com semáforos fechados, independentemente se o processo em questão vá ou não executá-las.

Para ser mais claro, o comando **oswais()** usado para T_i se bloquear não representa a situação em que T_i solicita a entrada da região crítica guardada por sem', mas sim, o ato de "pendurar" o processo T_i na fila de processos bloqueados do semáforo sem'.

(II) - *Antes de sair da região crítica*

Seja T_i o processo que possui prioridade $prio_i$ no momento de solicitar a saída da região crítica guardada por s_i.

O teste de saída da região crítica é idêntico ao do protocolo de herança de prioridade. Vide o item correspondente e a figura 3.5

3.2.2.4. Lemas e Teoremas

Lema 3

Seja T_B o processo de prioridade menor que a do processo T; então T pode ser bloqueado no máximo pelo equivalente à duração de uma região crítica do processo T_B.

Prova

A justificativa deste lema é análoga à do lema1.

Teorema 3

Um processo T pode ser bloqueado no máximo pelo equivalente à duração de uma região crítica de qualquer processo de prioridade mais baixa.

Prova

Suponha que o processo T possa ser bloqueado pelo equivalente à duração de n regiões críticas dos processos de prioridades menores. Nesse caso, segundo o lema 3, o processso T é bloqueado pelos n processos diferentes. Sejam T_1 e T_2 os dois primeiros processos de prioridade menor que bloquearam T; para que isso tenha ocorrido, os processos T_1 e T_2 deveriam estar nas suas regiões críticas quando T chegou. Suponha que T_2 entrou primeiro na região crítica guardada pelo semáforo S e que a prioridade topo de S é $ceil_S$. Assim, para T_1 entrar na sua região crítica, a sua prioridade tem que ser estritamente maior que $ceil_S$. Como foi assumido que T pode ser bloqueado por T_2 então, segundo o protocolo de prioridade topo, o valor de $ceil_S$ deve ser igual ou maior que a prioridade de T que, por sua vez, é maior que a prioridade de T_1 (Lembrando que $ceil_S$ é definido com a maior prioridade dos processos que usam o semáforo S). Portanto, T_1 não consegue entrar na sua região crítica enquanto T_2 estiver na dele. Isto contradiz a suposição inicial; logo, o teorema se justifica.

Observação

Uma vez que este protocolo garante que qualquer processo pode ser bloqueado apenas pelo tempo equivalente à duração de uma região crítica de qualquer outro processo de prioridade mais baixa, ele leva vantagem em relação ao protocolo de herança de prioridade o qual dá a duração máxima de bloqueio como sendo à de n regiões críticas.

Além disso, esse teorema contribui com o cálculo de uma condição suficiente para uma melhor avaliação do algoritmo de escalonamento "Taxa Monotônica".

Teorema 4

Um conjunto de n processos periódicos usando o protocolo de prioridade topo pode ser escalonado pelo algoritmo "Taxa Monotônica" se as seguintes condições são satisfeitas :

$$\frac{c_1}{p_1} + \frac{c_2}{p_2} + ,,, + \frac{c_n}{p_n} + \text{MAX}\left(\frac{B_i}{p_i}\right) \leq n * (2^{1/n} - 1)$$

$$i = 1 \text{ a } (n-1)$$

onde **n** é o número total de processos

B_i é o tempo de bloqueio no pior caso do processo T_i

$prio(T_1) > prio(T_2) > ,,, > prio(T_n)$

Observação

Para o valor de B_i é considerado o de pior caso. Segundo o teorema 3, um processo T_i pode ser bloqueado pela duração de uma região crítica de qualquer processo menos prioritário, sendo assim, é atribuído a B_i a duração da maior região crítica de todos os processos T_j , onde j=i+1 a n.

Deve ser lembrado que a condição suficiente, vista no item sobre processos periódicos e independentes deste capítulo, através da qual a viabilidade do algoritmo "Taxa Monotônica" é verificada para um conjunto de processos, é dada pela expressão abaixo :

$$\frac{c_1}{p_1} + ,,, + \frac{c_n}{p_n} \leq n * (2^{1/n} - 1)$$

Para os processos dependentes que carregam problemas de bloqueio, o incremento do fator B_i no cálculo parece ser razoável. Portanto, um conjunto de processos que obedeçam a essa condição pode ser escalonado cumprindo suas restrições de temporização.

3.3. Processos Periódicos e Aperiódicos Independentes (AMB 3)

3.3.1. Definição do Problema

Quando um sistema de tempo real com temporização rígida (Hard Real Time System) envolve não apenas processos periódicos mas também processos aperiódicos, o escalonador deve, através de uma série de algoritmos, atingir algumas metas importantes, tais como :

(**) Providenciar tempo de resposta o mais rápido possível para os processos que chegam ao sistema aleatoriamente, ou seja, os processos aperiódicos que não possuem prazos rígidos.

(**) Garantir também as restrições de temporização dos processos periódicos como,por exemplo, encontrar seus prazos sem atrasos e começar a execução respeitando seu "start time" etc.

Um meio de alcançar os objetivos citados acima é conceber um algoritmo de escalonamento que assegure o cumprimento, pelos processos periódicos, das suas restrições de temporização e criar um servidor que atenda às chegadas dos processos aperiódicos de modo o mais eficiente possível.

Três algoritmos de servidor de processos aperiódicos - Background, Polling e Deferrable Server [SPR 89] - são analisados em seguida.

3.3.2. Servidor Background

3.3.2.1. Definição do Servidor Background

O servidor Background atende os requisitos de execução dos processos aperiódicos, quando o processador estiver ocioso, ou seja, quando nenhum processo periódico desejar ser executado no momento.

(Exemplo VII)

Sejam A e B dois processos periódicos, e C e D, dois processos aperiódicos no sistema. Digamos que C e D cheguem nos instantes 5 e 12,

respectivamente, e as prioridades de A e B sejam determinadas de acordo com o algoritmo de taxa monotônica. Os dados são mostrados na tabela 3.1 :

proc	periodi-cidade	Período	Prazo	Priorid.	tempo de exec.	inst. de chegada
A	p.	10	10	alta	4	—
B	p.	20	20	baixa	8	—
C	ap.	—	—	—	1	5
D	ap.	—	—	—	0.5	12

tabela 3.1 - Dados dos Processos do Exemplo VII

Segundo a definição da taxa monotônica, ao processo de período menor é atribuído a maior prioridade; neste caso o processo A tem seu período igual a 10, menor que o de B, logo A é mais prioritário que B.

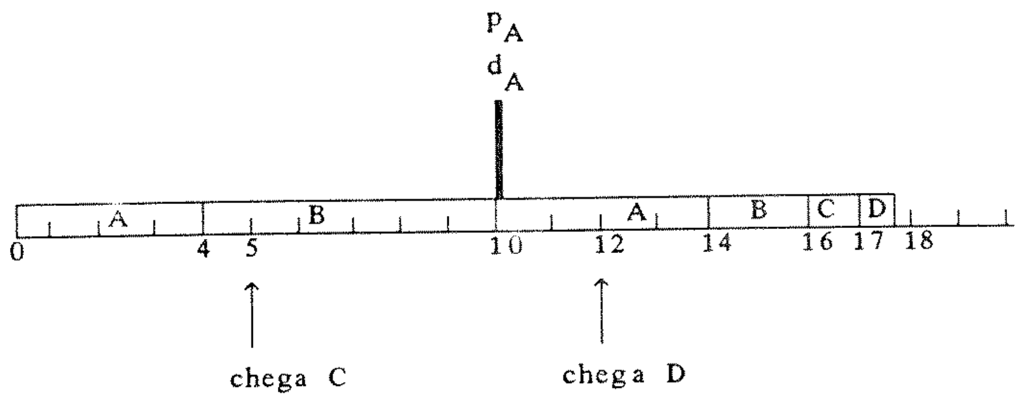


figura 3.12 - Exemplo do Servidor Background

No instante 0, A e B estão prontos mas A é escalonado para ser executado por ter prioridade maior. O processo A termina sua execução no instante 4 e o processo B toma a CPU. Porém, B é interrompido no instante 10 e cede a CPU ao

processo mais prioritário A pois este agora está novamente pronto. B retoma a sua execução logo depois que A libera a CPU no instante 14. No instante 16, B finaliza a execução e a CPU está desocupada. Observe-se que C e D chegaram durante as execuções dos processos periódicos A e B, mas não foram atendidos imediatamente. Somente quando nenhum processo periódico está pendente, C e D podem ser escalonados.

É obvio que quando o sistema contém uma carga grande de tempo de execução dos processos periódicos o atendimento dos aperiódicos fica atrasado indefinidamente. Além disso, o desempenho de tempo de resposta do servidor é bastante ruim. Porém, a idéia deste algoritmo é muito simples e sua implementação é trivial.

A implementação deste servidor é trivial pois basta atribuir aos processos aperiódicos as prioridades mais baixas do sistema (menor que a dos processos periódicos) de modo que só serão executados, quando nenhum processo periódico estiver solicitando execução, ou melhor, estiver na fila de prontos.

3.3.3. Servidor Polling

3.3.3.1. Definição do Servidor Polling

O servidor polling é criado como um processo periódico de alta prioridade e atende os processos aperiódicos pendentes em intervalos regulares. Se nenhum processo aperiódico está pendente, então o servidor fica suspenso até a ocorrência de seu próximo período, e durante esse tempo nenhum processo aperiódico é atendido. Assim sendo, os processos aperiódicos que chegarem logo após a suspensão do servidor devem esperar até o começo do próximo período do servidor polling. Portanto, o tempo de resposta médio dos processos aperiódicos com este algoritmo ainda é longo, apesar de ser mais eficiente que o background.

(Exemplo VIII)

Empregam-se os mesmos dados do (Exemplo VII) deste capítulo acrescentando-se o servidor como um processo periódico com tempo de execução (capacidade do mesmo) $c_s = 1$ e o período $p_s = 5$. A figura 3.13 mostra o funcionamento do servidor Polling neste exemplo.

UNICAMP
BIBLIOTECA CENTRAL

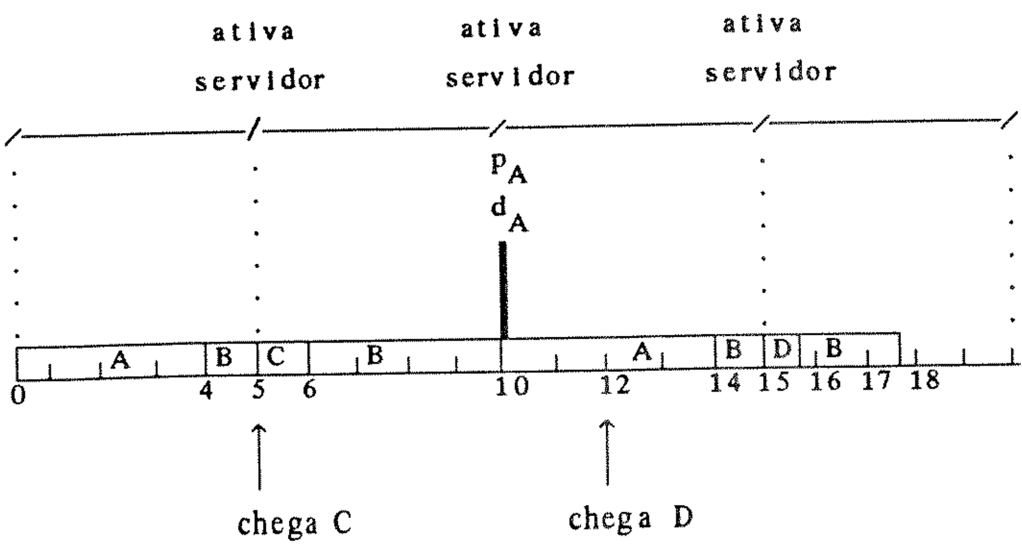


figura 3.13 - Exemplo do Servidor Polling

Observe-se que o servidor possui capacidade de processamento para atendimento de processos aperiódicos igual a 1.

No instante 5, chega o processo aperiódico C, quando o servidor também é ativado para o atendimento de aperiódicos. Assim, como C é o único processo pendente no momento, ele é executado. Ao terminar de executar C o processo periódico B continua sua execução.

No instante 10, o servidor polling verifica se houve novas chegadas de processos aperiódicos dentro do intervalo (5, 10); porém, nenhum aperiódico está pendente, por isso o servidor suspende-se e cede o seu tempo de execução aos periódicos. No instante 12, D chega durante a execução de A, contudo deve esperar (fica pendente) a reativação do servidor no próximo período.

No instante 15, o polling é ativado novamente e aloca um tempo de execução ao processo aperiódico pendente D. Note-se que o tempo de execução de D é igual a 0.5 e a capacidade do servidor é 1; então, depois de realizar o processo D, mesmo sobrando 0.5 de tempo da sua capacidade, o processo servidor dorme por não ter mais outro processo aperiódico para ser atendido.

Suponha que o tempo de execução de C seja 1.5 ao invés de 1, então o servidor ativado no instante 5 não suportaria a execução completa de C. Neste caso, o servidor apenas aloca 1 tempo de execução, que é a sua capacidade, para o processo C e o resto (0.5) do tempo de execução seria adiado para a próxima ativação do servidor

no instante 10.

3.3.3.2. Testes e Implementação

O processo servidor é criado como um processo periódico dotado com a prioridade mais alta de todos os demais processos periódicos e aperiódicos no sistema, sendo que no começo de cada período ele atende, dentro da sua capacidade, os processos aperiódicos que chegaram durante seu período anterior e que ainda não foram atendidos.

Do ponto de vista global, os processos aperiódicos possuem prioridades maiores que os periódicos. Porém, eles normalmente estão no estado dormente durante o funcionamento do sistema, exceto quando são atendidos pelo servidor. A tabela 3.3 mostra a hierarquia dos níveis de prioridade dos processos no sistema.

priori nível	tipo de processos	características principais
nível 1	servidor Polling	* É ativado em intervalos regulares (processo periódico) * No início do seu período verifica se algum processo está pendente. Se tiver, então executa-o, senão, finaliza a execução deste período * Atende os processos aperiódicos dentro da sua capacidade
nível 2	Processos Aperiódicos	* Inicialmente estão dormentes * São acordados pelo servidor no seu atendimento
nível 3	Processos Periódicos	* executam concorrentemente entre si de acordo com suas prioridades pré-fixadas

tabela 3.2 - Nível de Prioridades dos Processos e sua Descrição Funcional durante Teste do Servidor Polling

Proposta para implementar o Servidor Polling

Na verdade, a proposta de implementação para o servidor não se preocupa com a maneira pela qual os processos aperiódicos chegam e são recebidos no sistema. Todavia, alguns tratamentos adicionais devem ser feitos, durante as chegadas

dos aperiódicos, para colaborar com o processo servidor no atendimento.

A necessidade desses tratamentos surge pelo fato de que os processos aperiódicos não são atendidos imediatamente quando chegam ao sistema, mas aguardam a próxima ativação do servidor. Portanto, os tratamentos envolvem o armazenamento das informações dos processos aperiódicos e a suspensão dos mesmos, no instante de suas chegadas. Uma lista encadeada é proposta para este fim.

```
struct g
{
systag_t  idp;
unsigned short  falta;
struct g  *prox;
} guarda[APERIODIC]
```

idp ⟶ identificação do processo aperiódico

falta ⟶ indica o tempo de execução restante, quando o servidor esgotar a sua capacidade sem terminar a execução deste processo aperiódico. Quando o processo chega, este campo é preenchido com o valor do tempo total de execução.

prox ⟶ aponta para o próximo processo aperiódico

Esta lista é organizada com a ordem de chegada dos processos. Toda vez que o servidor é ativado, ela é usada para verificar a existência dos processos aperiódicos e, caso existam, atendê-los.

Uma vez que os processos aperiódicos são encontrados na lista guarda[], o servidor acorda-os, um de cada vez em função da ordem de chegada, usando o comando "Resume Process". Em seguida, o servidor deve ceder a CPU ao processo aperiódico e esperar pelo término de execução deste ou o fim da sua capacidade. Isto é realizado com o uso da primitiva de comunicação RECEIVE síncrona e temporizada pelo servidor e a SEND assíncrona no fim da execução de cada um dos processos aperiódicos. A função da RECEIVE temporizada é delimitar a execução dos aperiódicos dentro da capacidade do servidor, desde que o valor de temporização da RECEIVE é atribuído com o tamanho da sua capacidade (figura 3.14).

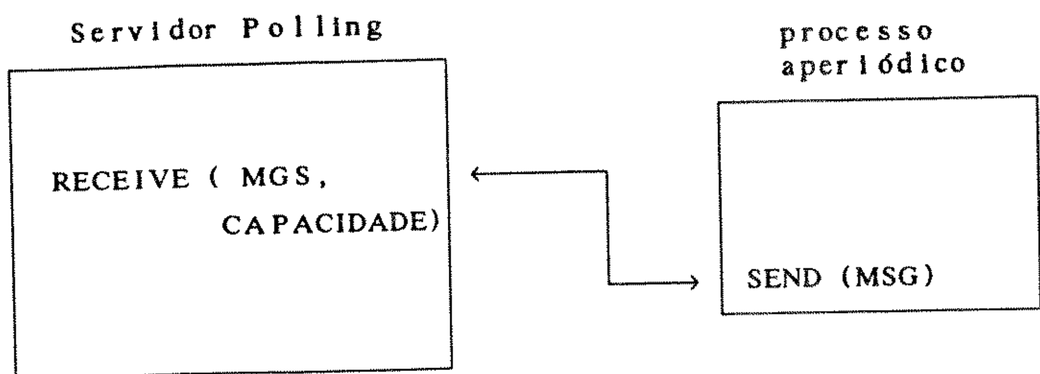


figura 3.14 - Atendimento Temporizado de um Processo Aperiódico pelo Servidor Polling

Assim, se a execução do processo aperiódico é menor que a capacidade do servidor então a SEND assíncrona é executada pelo processo aperiódico, antes de esgotar a temporização da RECEIVE síncrona, e o processo servidor retoma a execução da CPU. Porém, o servidor continua atendendo os outros processos aperiódicos, caso existam, até que a lista de aperiódicos fique vazia ou a sua capacidade tiver sido esgotada. Quando a segunda situação ocorrer, o servidor salva o tempo de execução restante para o campo **falta** do registro correspondente ao processo aperiódico que se encontra em execução.

É preciso ressaltar que o comando da primitiva RECEIVE síncrona e temporizada, usado no algoritmo, deve ser capaz de indicar se ele foi retornado com a execução da primitiva SEND correspondente ou se retornou com a temporização esgotada sem receber mensagem da primitiva SEND assíncrona esperada.

Algoritmo do processo Servidor Polling

Como o teste do algoritmo proposto foi feito sobre o núcleo DEADMOSI, o qual se encarrega ele mesmo de salvar e atualizar os dados dos processos periódicos, a ativação periódica dos processos periódicos é ignorada na elaboração de seus códigos.

Na verdade, o processo aperiódico deveria executar com a prioridade do servidor, mas para ter uma visão mais clara sobre o funcionamento global do algoritmo, criou-se então (vide tabela 3.2), para os aperiódicos, um nível de

prioridade 2 que está imediatamente abaixo do servidor e acima dos periódicos. Assim, essa abordagem não afetará o funcionamento do algoritmo.

passo1) Atualizar a capacidade **capacidade** do servidor com o valor máximo (abastecimento do servidor em cada período)

passo2) Verificar se existe algum processo aperiódico pendente na lista. Caso afirmativo, vá para passo3, senão, vá para passo5.

passo3) Acordar o processo aperiódico que será atendido pelo servidor

Comando RESUME PROCESS

osrespr(aperiod->idp)

Ceder a CPU para o processo aperiódico por um tempo limite que é o valor da capacidade disponível dessa instância do processo servidor.

Comando RECEIVE Síncrono

retorno = osrcvms(mail, capacidade, armaz, 5, tam)

Se **retorno** indicar o estouro de tempo limite do servidor antes de receber a mensagem da SEND indicando o término do processo aperiódico, então vá para passo4; senão, continua

Como o processo aperiódico já terminou sua execução, sem gastar toda capacidade do servidor, então sua exclusão da fila de prontos será automática pelo núcleo.

A capacidade **capacidade** é atualizada descontando o tempo de execução do processo aperiódico já atendido.

capacidade -= aperiod->falta

Por último, retira o registro que contém as informações sobre o processo aperiódico atendido, da lista encadeada que contém os processos aguardando atendimento pelo servidor. Volte para passo2.

passo4) Calcular o tempo de execução restante do processo aperiódico e guardá-lo no campo **falta**

aperiod->falta -= capacidade

As informações do processo aperiódico continuam armazenadas na lista encadeada.

O processo servidor deve fazer o processo aperiódico dormente outra vez até a ativação do próximo período para continuar o atendimento do processo aperiódico.

Comando SUSPEND PROCESS

ossuspr(aperiod->idp)

passo5) Fim. Ficará dormente até o próximo período do processo servidor.

3.3.4. Servidor DS - Defrrable Server

Este servidor supera as desvantagens dos servidores Background e Polling pois providencia o atendimento imediato para os processos aperiódicos, assim que estes chegam. Logo, seu tempo médio de resposta apresenta uma melhoria em relação aos dois servidores anteriores.

3.3.4.1. Definição do Servidor DS

Assim como o algoritmo do servidor Polling, o algoritmo DS também cria um processo servidor para atender os aperiódicos. Porém, em qualquer instante no período do servidor DS, os processos aperiódicos podem ser atendidos até um limite de tempo correspondente à capacidade do servidor.

O limite da capacidade do servidor DS deve ser respeitado. Na ativação de cada novo período sua capacidade é reabastecida ao máximo e pode ser gasta gradualmente no decorrer do mesmo.

Resumindo então as idéias e destacando as diferenças entre os servidores Polling e DS tem-se a seguinte observação na tabela 3.3.

Depois de reabastecer a capacidade do servidor no início do seu período :	
Polling	Polling executa, caso existam processos aperiódicos na lista encadeada de processos chegados. Quando terminar a execução do último processo aperiódico da lista,ou esgotar a sua capacidade, o servidor não mais oferece atendimento até o próximo período
DS	DS possui uma capacidade máxima e somente gasta-a quando o processo aperiódico chega durante o período atual. Assim, o atendimento dos aperiódicos torna-se bem mais flexível

tabela 3.3 - Principais Diferenças entre Servidor Polling e DS

(Exemplo IX)

Usam-se os mesmos dados da tabela 3.1 acrescentando-se o servidor como um processo periódico com tempo de execução c_s= 0.8 (sua capacidade) e o período p_s= 5. O funcionamento do servidor DS neste exemplo será ilustrado na figura 3.15.

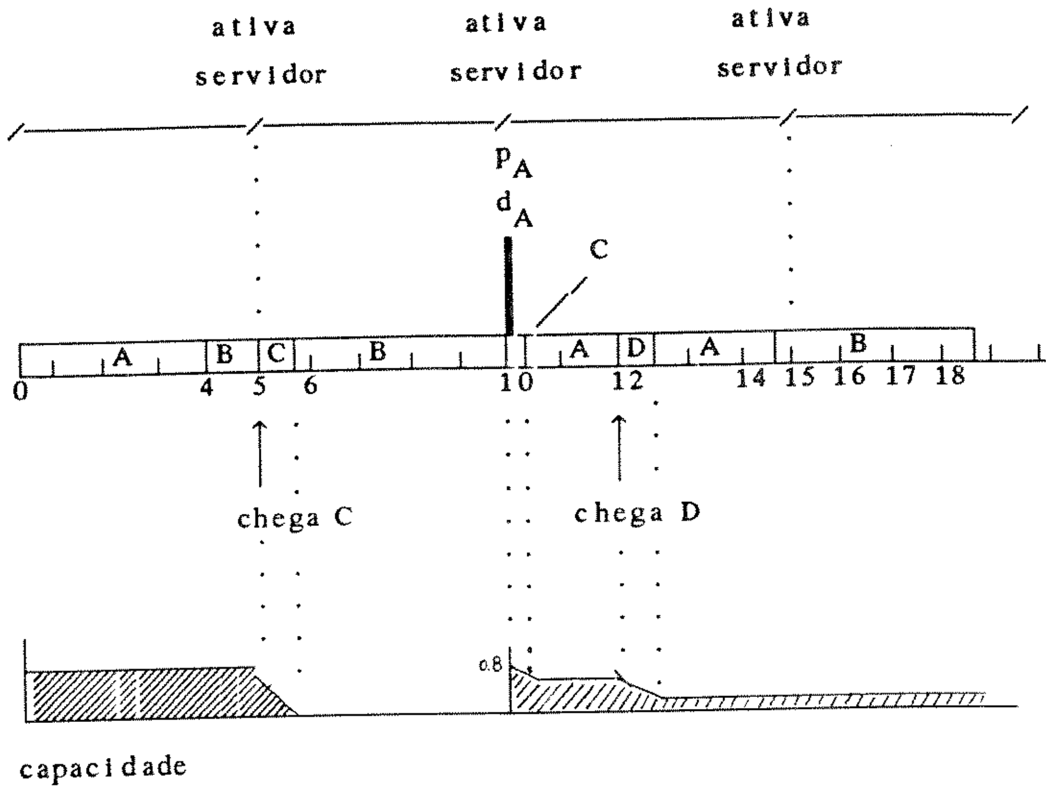


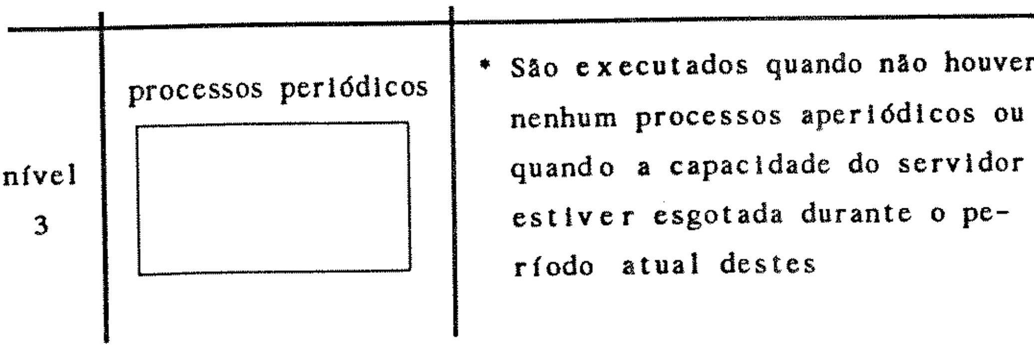
figura 3.15 - Exemplo do Servidor DS

Inicialmente a capacidade do servidor é máxima. Durante o intervalo (0,5), não chega nenhum processo aperiódico. No instante 5, o servidor reabastece de novo e nesse momento chega o processo C, então C é atendido. Como o tempo de execução de C é 1 e a capacidade do servidor é 0.8, a capacidade acaba no período atual, deixando como sobra 0.2 de tempo de C não executado. Assim, quando o servidor é reativado no instante 10, a execução de C se completa. Agora, a capacidade disponível ao servidor DS no período (10,15) é 0.6. Antes de recarregar a capacidade, ou seja, vencer o período do servidor, no instante 12, o processo D chega e é logo atendido. O tempo de execução de D é 0.5, a capacidade atual é 0.6, então D é executado até o fim, ainda sobrando 0.1 de capacidade do servidor. No instante 15 quando ocorre a nova instância do servidor, a capacidade torna a ter valor máximo, igual a 0.8

3.3.4.2. Testes e Implementação

O processo servidor DS possui, como o servidor Polling, prioridade maior que a dos processos aperiódicos que por sua vez é maior que a dos periódicos. A tabela 3.4 mostra uma visão global sobre os níveis de prioridade dos processos no sistema e as características principais de cada um destes.

priori nível	tipo de processos	características principais
nível 1	servidor DS	* repete periodicamente * Toda vez que ele é ativado verifica se há algum processo pendente que não foi completado durante o atendimento no período anterior * Se não houver chegado nenhum processo aperiódico, então o servidor fica esperando a mensagem de chegada dos processos
nível 2	processos aperiódicos	* Inicialmente estão dormentes * São acordados pelo servidor no seu atendimento * Quando estes chegam, a rotina de tratamento de chegadas deve avisar o servidor * Uma vez que o servidor é avisado estes são atendidos por ele considerando sua capacidade



nível 3	processos periódicos	* São executados quando não houver nenhum processos aperiódicos ou quando a capacidade do servidor estiver esgotada durante o período atual destes

tabela 3.4 - Nível de Prioridades dos Processos e sua Descrição Funcional durante Teste do Servidor DS

Proposta para Implementar o Servidor DS

Como foi citado no ítem anterior, do servidor Polling, certos tratamentos são importantes para cooperar com o funcionamento do algoritmo proposto.

Pela definição deste servidor, os processos aperiódicos devem ser atendidos dentro de certas condições, assim que chegam ao sistema. Portanto, o servidor deve ser avisado imediatamente das chegadas dos processos aperiódicos. Assim, é sugerido o uso do comando SEND assíncrono na rotina de tratamento de chegadas (ou qualquer outro mecanismo do gênero) para enviar a mensagem de chegada ao servidor.

O outro tratamento indispensável é o armazenamento de informação dos processos chegados na lista encadeada, **guarda[]**, a qual apresenta a mesma estrutura de dados descrita no ítem do servidor Polling.

Quando o servidor DS é ativado, ele detecta, caso existam, os processos aperiódicos chegados e que não tiveram efetivado o seu atendimento do período anterior, e executa-os.

Se não houver nenhum processo aperiódico acumulado da sua instância anterior, o servidor DS usa o comando RECEIVE síncrono e temporizado, **Rec1**, para aguardar o aviso da chegada de algum processo aperiódico. Note que **Rec1** é o comando de comunicação correspondente com o SEND assíncrono, **Sen1**, da rotina de tratamento de chegadas. Nesse caso, é atribuído à temporização de **Rec1** o tempo restante para o vencimento do período atual do servidor. Assim, se nenhum processo aperiódico chega durante a espera (temporização de **Rec1**) então o servidor finaliza a execução da instância atual e inicia outro período.

Caso chegue um processo aperiódico, o modo de atendimento é idêntico ao do Polling, usando também outro comando RECEIVE síncrono e temporizado ,Rec2, para delimitar a execução do aperiódico dentro da sua capacidade disponível. E, logicamente, todos os processos aperiódicos devem incluir o comando SEND assíncrono, Sen2, no fim do seu código. Porém, existe uma situação especial onde um processo aperiódico chega quase no fim do período atual do servidor e a capacidade do mesmo foi pouco gasta, pela chegada escassa de processos aperiódicos, durante o período. Em outras palavras, o valor da capacidade do servidor é maior que o tempo restante para o vencimento do período. Assim, uma verificação sobre a capacidade restante do servidor em relação ao tempo restante do período do mesmo deve ser feita antes de ceder a CPU para o aperiódico.

Depois de atender algum processo aperiódico o servidor atualiza o valor da sua capacidade e o tempo restante do seu período e usa novamente Rec1 para continuar à espera de novos processos aperiódicos, até o esgotamento do período, se for o caso. Assim, são necessários dois comandos RECEIVE síncronos para implementar o servidor DS (figura 3.16).

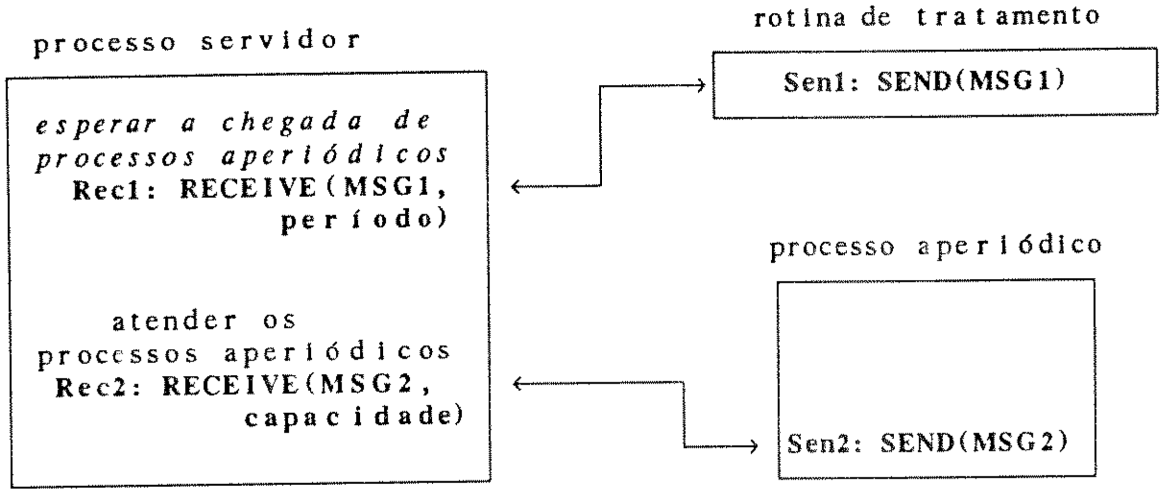


figura 3.16 - Atendimento e Espera de Chegadas Temporizados de um Processo Aperiódico pelo Servidor DS

Algoritmo do processo Servidor DS

passo1) Inicializar **"capacidade"** com a capacidade máxima do servidor e **"limite"**, o tempo para a espera de processos chegados, com o tamanho do período do servidor

passo2) Verificar se existe na lista algum processo aperiódico pendente. Caso tenha, então vá para passo3; senão, vá para passo4.

passo3) Acordar o processo aperiódico o qual será atendido pelo servidor

Comando RESUME PROCESS

osrespr(aperiod->idp)

Ceder a CPU para o processo aperiódico por um tempo limite que é o valor da capacidade disponível dessa instância do processo servidor ou o tempo restante do período atual, o que for menor dos dois.

templim = Min(capacidade, limite)

Comando RECEIVE Síncrono

retorno = osrcvms(mail2, templim)

Se **retorno** indicar o estouro da sua capacidade ou do seu período antes de receber a mensagem da SEND indicando o término do processo aperiódico, então vá para passo5 senão continua

Se **retorno** indicar que o processo aperiódico terminou sua execução, sem gastar toda capacidade do servidor, então sua exclusão pelo núcleo da fila de prontos será automática pelo núcleo.

A capacidade **capacidade** é atualizada descontando o tempo de execução do processo aperiódico já atendido.

capacidade -= aperiod->falta

O tempo **limite**, com o qual o servidor deve esperar, na instância atual, as chegadas dos aperiódicos é atualizado

Retirar o registro que contém as informações sobre o processo aperiódico atendido, da lista encadeada, **guarda[]**, que contém os processos que estão aguardando pelo atendimento do servidor. Vá para passo2.

passo4) Ceder a CPU para os processos periódicos e ficar esperando dentro de um tempo limite,**limite**, a mensagem de chegadas dos processos aperiódicos

Comando RECEIVE Síncrono

retorno = osrcvms(mail1, limite)

Se **retorno** indicar que nenhum processo aperiódico chegou até fim do seu período então vá para passo6, senão (chegou algum processo aperiódico no sistema) volte para passo3.

passo5) Calcular o tempo de execução restante do processo aperiódico e guardá-lo no campo **falta**

aperiod->falta -= templim

As informações do processo aperiódico continuam armazenadas na lista encadeada.

O processo servidor deve fazer o processo aperiódico dormir até a ativação do próximo período para continuar o atendimento do processo aperiódico.

Comando SUSPEND PROCESS

ossuspr(aperiod->idp)

passo6) Fim. Espera a próxima ativação.

CAPÍTULO 4

ESCALONADOR DINÂMICO "EARLIEST DEADLINE"

4.1. Processos periódicos e independentes - AMB 4

4.2. Processos periódicos e dependentes

4.2.1. Rendezvous determinístico - AMB 5

4.2.2. Monitor "Kernelized" - AMB 6

4. ESCALONADOR DINÂMICO "EARLIEST DEADLINE"

O escalonador "Earliest Deadline" adota uma filosofia de escalonamento dinâmico, onde as prioridades dos processos são determinadas dinamicamente no decorrer do sistema, de acordo com o grau de urgência para cumprirem seus prazos (deadlines).

Aqui, "Prioridade Dirigida" e "Preempção" continuam sendo as principais características do escalonador.

Neste capítulo, as análises estão centradas nas técnicas usadas para resolver problemas ocorridos nos ambientes AMB4, AMB5 e AMB6, dando ênfase à discussão sobre a insuficiência e falha das mesmas.

4.1. PROCESSOS PERIÓDICOS E INDEPENDENTES (AMB 4)

Os processos periódicos são aqueles que requerem execução (ou que ficam prontos) em cada intervalos constantes, definidos como seus períodos. Assim, o período de um processo é considerado como um dos fatores mais importantes na restrição de temporização em Hard Real Time System, pois ele constitui um prazo de execução da instância atual e o vencimento dele deve ser indesejado. A independência entre os processos ocorre, quando nenhum destes possui relação de comunicação ou de compartilhamento de recursos comuns com nenhum outro processo.

(Exemplo X)

Sejam A, B e C três processos independentes e periódicos e cada um deles possuindo tempo de execução e período próprios. Suponha que os prazos deles tenham os mesmos valores de seus períodos.

	tempo de execução	período	prazo
A	2	5	5
B	1	4	4
C	1	3	3

tabela 4.1 - Dados dos Processoss do Exemplo X

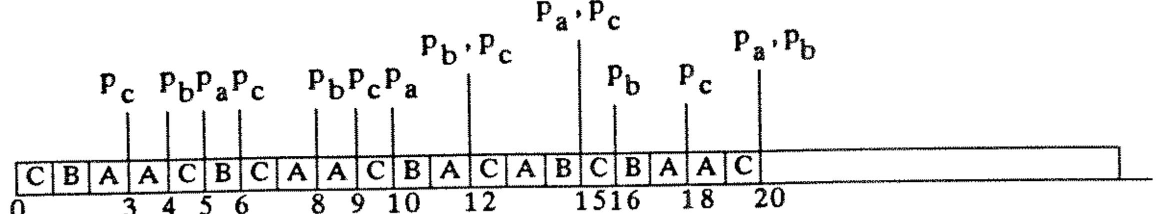


figura 4.1 - Escalonador "Earliest Deadline" no Exemplo X

(Exemplo XI)

Se aumentarmos o tempo de execução do processo B então :

	tempo de execução	período	prazo
A	2	5	5
B	2	4	4
C	1	3	3

tabela 4.2 - Dados dos Processos do Exemplo XI

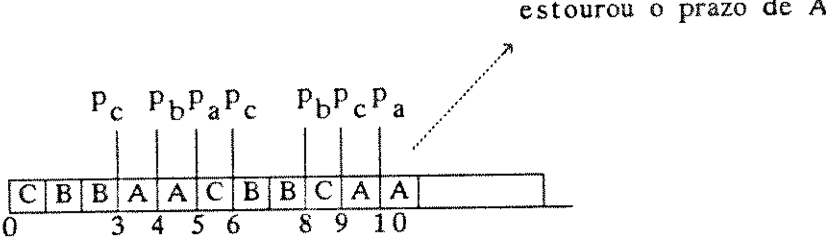


figura 4.2 - Vencimento de Prazo do Processo A - Exemplo XI

Desse modo, o algoritmo "Earliest Deadline" apresenta um escalonamento de processos viável somente sob uma condição necessária e suficiente [LIU 73] que será mostrada a seguir.

Teorema

Para um dado conjunto de **m** processos periódicos e independentes, onde o valor do prazo de cada processo é igual ao de seu período, o algoritmo de escalonamento "Earliest Deadline" é viável, no caso de monoprocessamento, se e somente se

$$(c_1 / p_1) + (c_2 / p_2) + ,,, + (c_m / p_m) \leq 1$$

onde p_i é o período do processo T_i e c_i o tempo de execução.

Prova

Seja C = { T_1 ,,, T_m} um conjunto de processos periódicos cada um dos quais, T_i, possui parâmetros c_i , tempo de execução, d_i , seu prazo e p_i , seu período.

Toma-se **P** = (p_1 * p_2 * ,,, * p_m), múltiplo comum dos períodos de todos os processos, como sendo um intervalo para esta análise. Assim, o número de ocorrências de T_1 dentro de **P** é (p_2 * p_3 * ,,, * p_m) e T_2 , (p_1 * p_3 * ,,, * p_m), assim por diante. Logo, o tempo total de execução do processo T_1 dentro de **P** será calculado como (c_1* (p_2 * p_3 * ,,, * p_m)) e o de T_2 (c_2* (p_1 * p_3 * ,,, * p_m)), etc. Portanto, o somatório de tempo total de todos os processos não deve ser maior que o tamanho do intervalo de tempo **P**, de modo que nele se encaixem todas as execuções dos processos do sistema.

$$(c_1 * (p_2 * p_3 * ,,, *p_m)) + (c_2 * (p_1 * p_3 * ,,, * p_m)) + ,,, +$$

$$(c_m * (p_1 * p_2 * ,,, * p_{m-1})) \leq (p_1 * p_2 * ,,, * p_m)$$

Simplificando a equação (dividindo cada lado por **P**) tem-se :

$$(c_1 / p_1) + (c_2 / p_2) + ,,, + (c_m / p_m) \leq 1$$

No **(Exemplo X)** a soma de (c_i / p_i) tem valor ((1/3)+(1/4)+(2/5)) =

59/60 menor que 1, satisfazendo a condição; logo, o escalonador é viável para esse conjunto de processos. Já no (Exemplo XI), a soma total é ((2/5)+(2/4)+(1/3))= 74/60,o que é maior que 1 e, segundo o teorema, o escalonamento destes processos não é viável. O estouro de prazo do processo A no instante 10 confirma essa afirmação.

Observação

Segundo Liu [LIU 73] a escalonabilidade de um conjunto de processos independentes é garantida se o seu fator de utilização U é menor ou igual ao menor limite superior U_{MS}. Assim, podemos interpretar, a partir desta condição, que o menor limite superior de todos os conjuntos de processos (independe de seu tamanho) escalonados pelo "Earliest Deadline", é uniformemente 100%

4.2. Processos Periódicos e Dependentes

O problema em cumprir os requisitos de restrição de temporização complica-se, quando os processos interagem através de um mecanismo de coordenação interprocessos. É muito comum, em sistemas de tempo real, as situações em que os processos compartilham uma área comum ou trocam mensagens entre si. O compartilhamento de recursos é geralmente realizado, usando uma região crítica guardada por semáforos. No caso, a dependência entre dois ou mais processos surge, quando um processo toma a região crítica e outros, que precisam do mesmo recurso no momento, são obrigados a esperar que saia da mesma.

Os processos que trocam mensagens usam geralmente as primitivas SEND e RECEIVE, síncronas ou assíncronas, para efetuar comunicação interprocessos. O envio de mensagens assíncronas não estabelece nenhuma relação com outro processo pois não é necessário a espera da confirmação do recebimento pelo destinatário. Já o envio síncrono determina uma relação de dependência entre os dois processos comunicantes, uma vez que o recebimento da mensagem é sempre suposto um ato síncrono.

Para tal ambiente, é proposto um modelo básico de processos a fim de facilitar o desenvolvimento de algoritmos que tentarão dar soluções ao sistema, onde as tarefas possuem relações de dependência.

Cada processo T_i consistirá de uma cadeia de blocos de escalonamento, { $T_{i,j}$, j = 1 a n } onde $T_{i,j}$ é um pedaço de código de programação a ser executado após o bloco $T_{i,j-1}$, e n é o número de blocos contidos no processo.

Cada bloco $T_{i,j}$ tem um tempo de execução $C_{i,j}$ cujo valor é conhecido a priori (fornecido pelo projetista durante a inicialização do sistema). Observando que

$$\sum_{j=1}^{n} C_{i,j} = C_i ,$$

onde C_i é o tempo de execução total do processo T_i

Os blocos de escalonamento de um processo são separados em pontos, onde a coordenação interprocessos ocorre, ou seja, as primitivas de comunicação devem aparecer somente entre os blocos.

4.2.1. Rendezvous Determinístico (AMB 5)

4.2.1.1. Definição do Problema

O rendezvous determinístico é um mecanismo de comunicação entre processos, através do qual estes trocam informações entre si. Neste caso, o envio e recebimento das mensagens é síncrono. Em outras palavras, quando o processo T_i tenta executar a primitiva de rendezvous ele deve esperar até que o processo relacionado T_j também execute a sua primitiva correspondente. Diz-se então que T_i e T_j são processos comunicantes.

Normalmente, os tempos gastos pela manipulação das primitivas de rendezvous (overhead) são incluídos dentro do tempo de execução dos blocos de escalonamento. O modo como será estimado o tempo de execução de cada processo e de cada bloco não será discutido neste trabalho; cabe ao usuário tratar desse assunto, tendo em mente que a estimativa e a distribuição do tempo de overhead do mecanismo são necessárias para que o sistema possa ter um controle de tempo mais exato.

Deve-se destacar que o mecanismo de rendezvous não permite a comunicação entre processos periódicos e aperiódicos. Um processo periódico é, por definição, executado regularmente uma vez a cada determinado intervalo de tempo; já os processos aperiódicos não possuem tal característica; assim, para cada execução da primitiva de solicitação de comunicação (rendezvous) pelo processo periódico não se pode garantir a execução da primitiva correspondente pelo processo aperiódico, antes do estabelecimento do próximo período

Assim, se dois processos periódicos querem se comunicar, uma

condição de compatibilidade é imposta.

Dois processos são ditos compatíveis se possuem o mesmo período ou se o período de um é múltiplo exato do período do outro. Caso a condição acima não se verifique, então a não correspondência de ocorrências de um processo em relação ao outro leva à perda do controle na comunicação. Por exemplo : suponhamos dois processos, T_1 e T_2 , que se comunicam num determinado ponto do corpo de execução, sendo o período de T_1, P_1 = 2, e o de T_2, P_2= 3. Observe que dentro do intervalo [0,6], T_1 deve ocorrer três vezes e T_2, duas vezes. Logo, a última instância de T_1 não conseguirá um rendezvous com T_2 .

Se o período do processo B, P_B, é n vezes o período do processo A, P_A , implica que A deve ocorrer n vezes para cada ocorrência de B, então o processo B precisa de (n*p) pontos de comunicação para suportar os (p) pontos de cada ocorrência do processo A. A figura 4.3 mostra a situação em que p=1, ou seja, processo A possui um único ponto de rendezvous com o processo B, e o processo B contém n pontos para troca de mensagens.

$$P_B = n * P_A$$

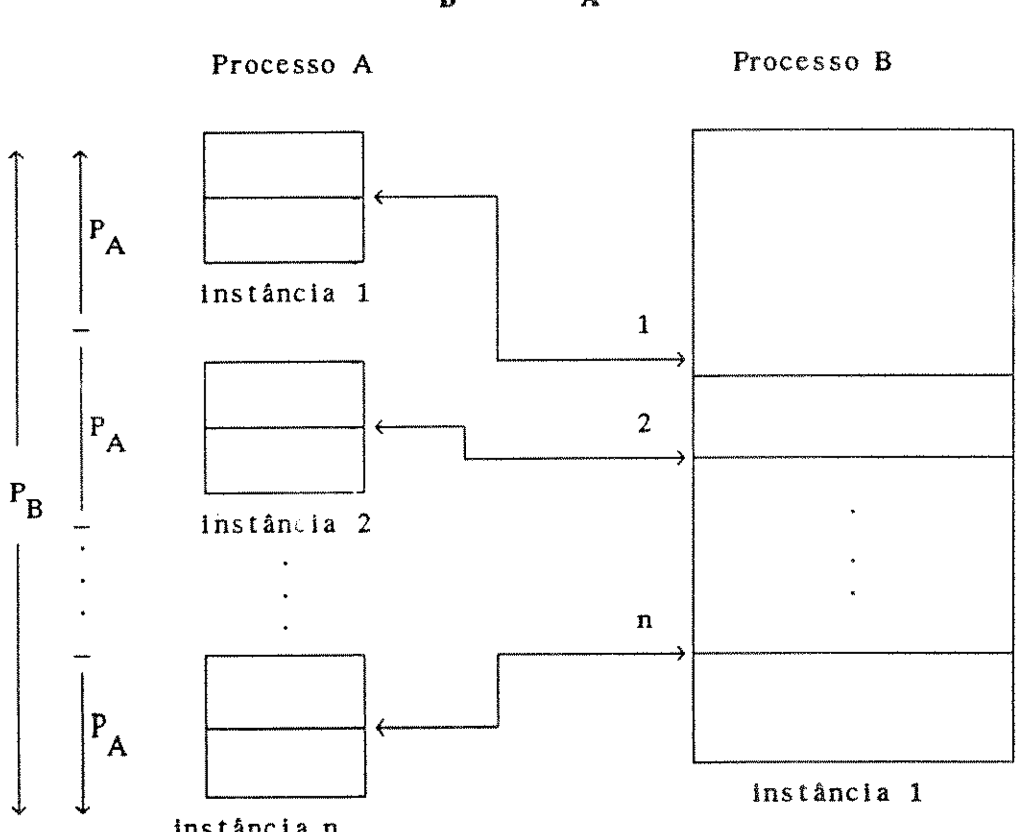


figura 4.3 - Pontos de Comunicação entre P_A e P_B

Se dois processos são comunicantes, através da transitividade, então eles também devem ser compatíveis. A figura 4.4 mostra essa propriedade :

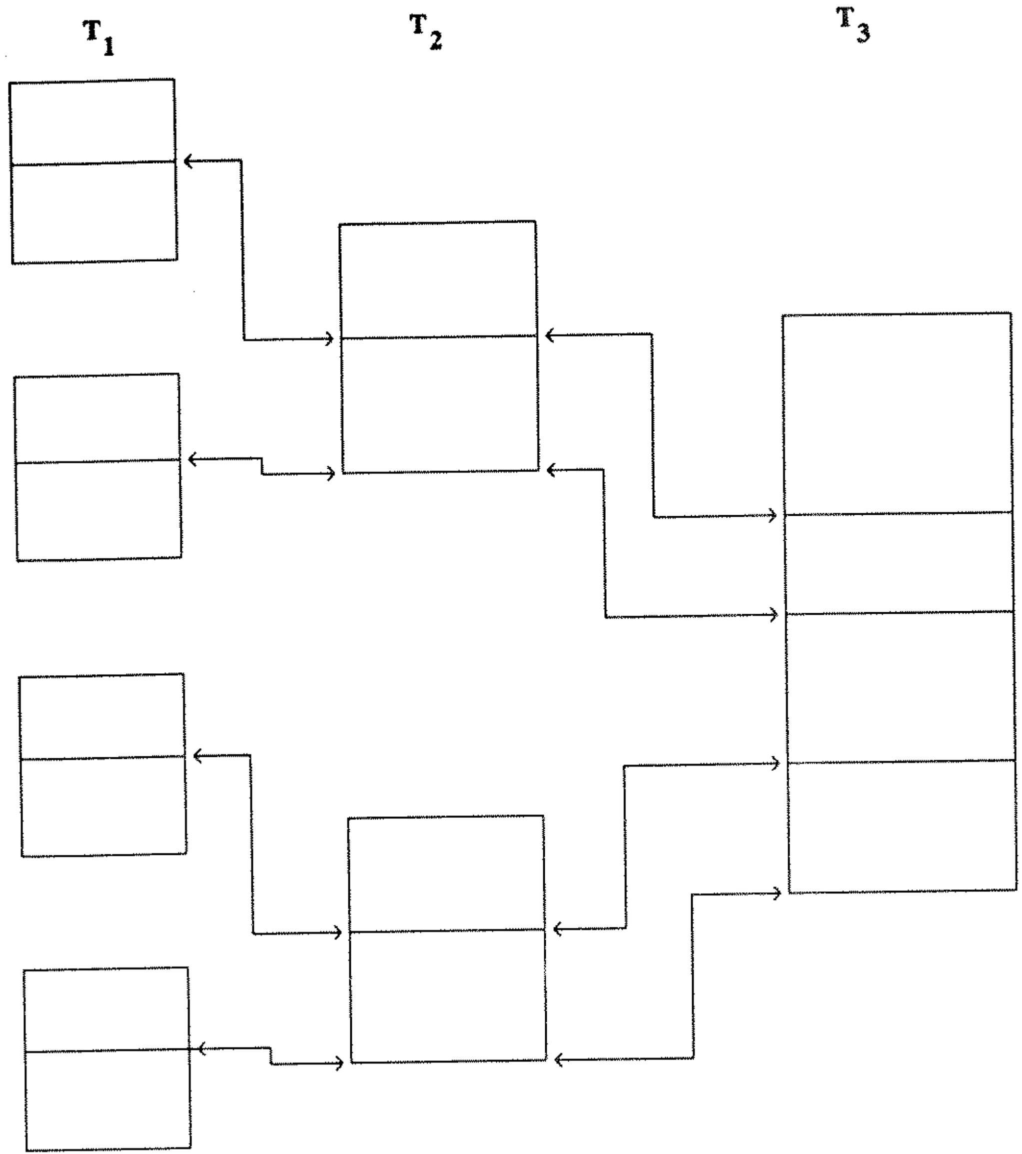


figura 4.4 - Propriedade de Transitividade entre dois Processos Comunicantes

Suponha que os pares T_1 e T_2 , T_2 e T_3 constituem relações comunicantes entre si e ainda que T_3 mantenha a mesma relação com T_1 transitivamente (via T_2), então T_1 e T_3 são compatíveis.

Portanto, pode-se dizer que os processos, que possuem relação comunicante direta ou transitiva entre si, pertencem à mesma classe de equivalência.

Além de serem compatíveis, os processos com relação comunicante estabelecem uma restrição de precedência. É óbvio que a natureza síncrona do mecanismo rendezvous força uma ordem de execução entre os blocos de escalonamento, devido às esperas obrigatórias para um envio ou recebimento das mensagens.

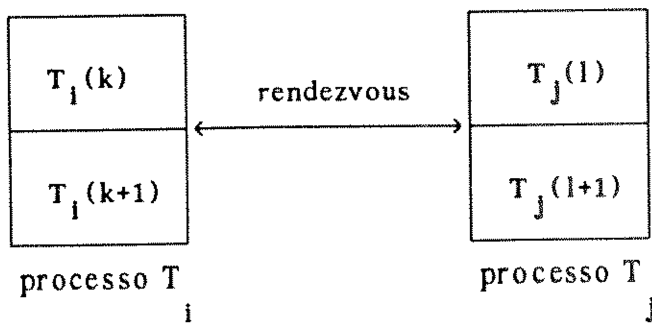


figura 4.5 - Relação de Precedência entre dois Processos Comunicantes

Na figura 4.5 dois processos T_i e T_j estão destinados a um ato de rendezvous, com as primitivas ocorrendo entre os blocos de escalonamento $T_i(k)$, $T_i(k+1)$ e $T_j(l)$, $T_j(l+1)$. Portanto, definem-se as relações de precedência interprocessos como :

$$T_i(k) \xrightarrow{\text{precede}} T_j(l+1)$$

$$T_j(l) \xrightarrow{\text{precede}} T_i(k+1)$$

e as relações de precedência intraprocessos são :

$$T_i(k) \xrightarrow{\text{precede}} T_i(k+1)$$

$$T_j(l) \xrightarrow{\text{precede}} T_j(l+1)$$

Agora, o problema de escalonamento com processos dependentes e periódicos deve ser enfrentado. A seguir mostrar-se-á que o escalonador "earliest

deadline", o qual adota a maior prioridade de execução para o processo que possui vencimento de prazo mais próximo e que não esteja bloqueado pelo rendezvous, deixa de ser ótimo quando o problema de precedência for levado em consideração no sistema.

(Exemplo XII)

Sejam T_1, T_2, T_3 três processos periódicos, sendo o processo T_1 dividido em dois blocos de escalonamento com seus tempos de execução : $c_{11} = c_{12} = 1$, prazo $d_1 = 3$ e período $p_1 = 5$; o processo T_2 também contém dois blocos de escalonamento : $c_{21} = 1$ e $c_{22} = 3$, seu prazo $d_2 = 10$, e seu período $p_2 = 10$. T_1 deve trocar mensagens com o T_2 depois de seu primeiro bloco, c_{11}.Como T_3 não se comunica com nenhum processo ele possui um único bloco : $c_3 = 1$ com prazo $d_3 = 9$ e período $p_3 = 10$. Esses dados podem ser organizados como mostrado na tabela 4.3, e a estrutura de relação interblocos ou interprocessos é mostrada na figura 4.6.

	T_1	T_2	T_3
d_i	3	10	9
p_i	5	10	10
c_i	$c_{11}=1$, $c_{12}=1$	$c_{21}=1$, $c_{22}=3$	$c_3 = 1$

tabela 4.3 - Dados dos Processos do Exemplo XII

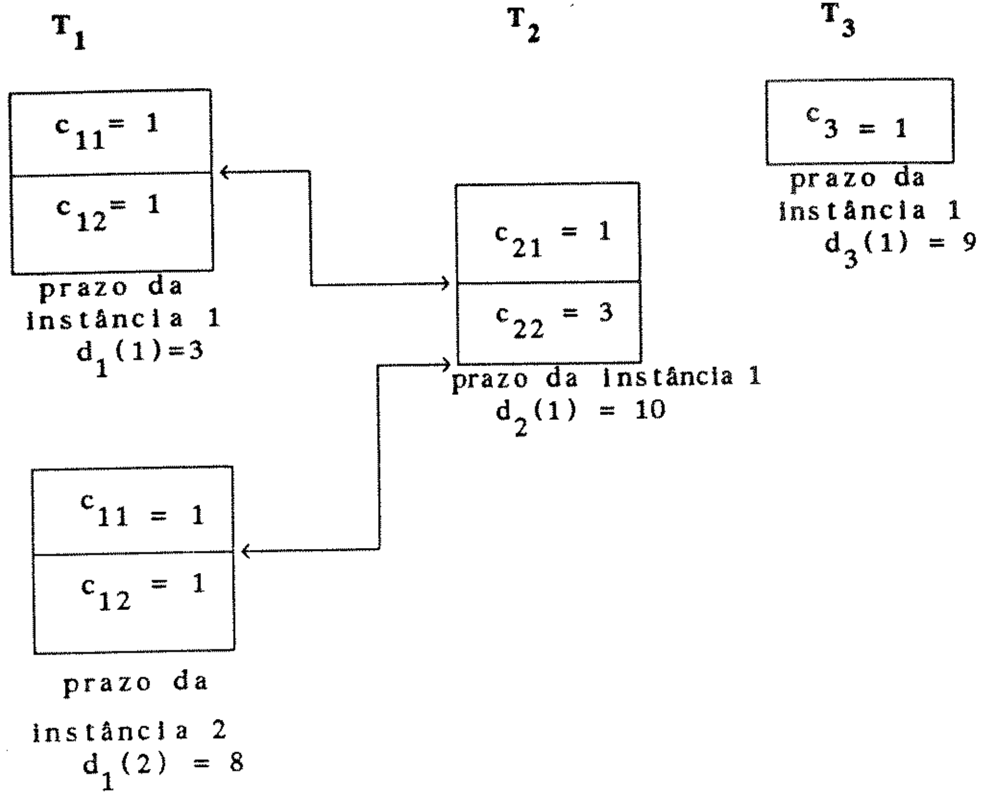


figura 4.6 - Comunicação Rendezvous entre Processos do Exemplo XII

Os dois comunicantes T_1 e T_2 são compatíveis pois o período de T_2, p_2=10, é exatamente duas vezes o período de T_1, p_1=5. E ainda, cada instância do processo T_2 contém dois pontos de rendezvous, enquanto cada instância de T_1 tem um único ponto de comunicação com T_2, o que é coerente segundo a análise do ítem anterior.

Aplicando o escalonador "Earliest Deadline" o aspecto global da execução dos processos do sistema é representado na figura 4.7 abaixo :

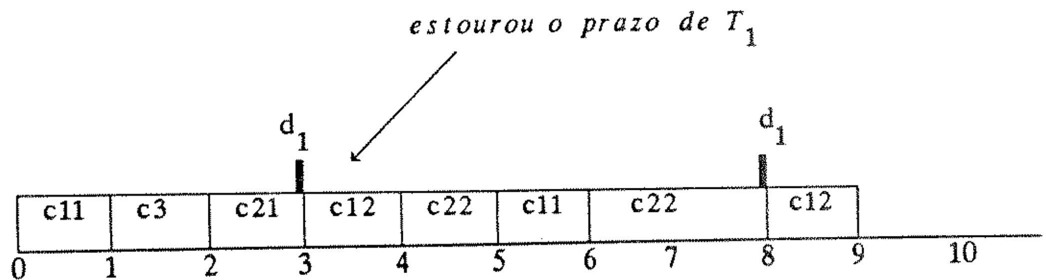


figura 4.7 - Vencimento de Prazo do Processo T_1 do Exemplo XII

No instante 0, o processo T_1 é escalonado pelo núcleo, por ter o prazo mais próximo (d_1= 3). Após a execução do primeiro bloco, c_{11}, o processo T_1 é bloqueado no instante 1, esperando o rendezvous correspondente; T_3 toma a execução da CPU, pois seu prazo, d_3= 9, é o mais próximo dentro dos processos prontos. Terminando a execução de T_3, e com T_1 ainda à espera de comunicação, T_2 assume a execução. No instante 3, T_2 responde ao rendezvous de T_1 ; este é liberado e deve continuar sua execução imediatamente com a maior prioridade. Porém, neste momento, seu prazo para finalizar a execução (d_1= 3) já está vencido.

A restrição de temporização é violada, o que resulta na impraticabilidade do uso desta técnica de escalonamento para esse caso.

Na verdade, o bloco c_{21} tem mais urgência de ser executado que o bloco c_3, apesar do prazo d_3= 9 ser o mais próximo naquele momento. Essa contradição deve-se ao fato da existência de um outro processo mais prioritário no sistema, T_1, que precisa urgentemente da execução do bloco c_{21}, pois deverá realizar um rendezvous com T_2 para prosseguir e terminar sua execução, antes do vencimento do seu prazo. Deve ser notado que a execução do bloco c_3 nessa situação poderia ser adiada sem afetar o desempenho do sistema.

Esse problema pode ser contornado se a técnica de "prazos revisados" for aplicada ao sistema.

4.2.1.2. Definição da Técnica de "Prazos Revisados"

A idéia principal desta técnica é ajustar os prazos de cada bloco de escalonamento, de acordo com o grau de urgência explícito e implícito, respectivamente, do seu prazo e da precedência exercida aos processos com prioridades mais altas. Uma base de dados, incluindo os prazos revisados de todos os blocos, é construída para a consulta dinâmica do escalonador no decorrer da execução do sistema.

Antes de descrever o algoritmo da técnica de "prazos revisados", alguns parâmetros e notações devem ser explicados.

Nesta técnica, é construído um grafo onde as relações de precedência interprocessos e intraprocessos são incorporadas e sobre o qual os prazos serão recalculados.

Seja **L** o mínimo múltiplo comum de todos os períodos no sistema dentro do qual os prazos de valor relativo serão verificados e reajustados quando necessário. Assim sendo, durante o funcionamento do sistema a reciclagem dos prazos em

cada L unidades de tempo é imprescindível.

Caso em um conjunto de processos exista somente processos da mesma classe de equivalência, então será atribuído a L o maior período de todos, visto que um sempre é igual ou múltiplo exato do outro. Porém, se alguns processos independentes estiverem contidos neste modelo, eles devem ser representados, no grafo referido, sob a forma de nós sem ligação nenhuma com outros.

Na verdade, os prazos destes processos não serão levados em consideração no ajuste; portanto, o algoritmo resolve apenas parcialmente o problema de escalonamento. Deixar-se-á esta discussão para uma análise posterior. A notação T_{ij} indica o j-ésimo bloco do processo T_i e $T_i(k)$, a k-ésima instância de T_i representando a k-ésima ocorrência ou repetição do processo periódico. E ainda, $S \longrightarrow S'$ tem como significado " o bloco de escalonamento S precedendo o bloco S' ".

4.2.1.2.1. Algoritmo da Técnica

passo1) Organizar os blocos de escalonamento de todos os processos dentro do intervalo [0,L] em ordem topológica, ou seja, formar uma sequência, respeitando as relações de precedência interprocessos e intraprocessos.passo2) Inicializar os prazos (deadline) da k-ésima instância do bloco T_{ij} com o valor

$$d_{ij}(k) = (k - 1) * p_i + d_i$$

no intervalo [0,L], onde p_i é o período do processo T_i e d_i , o prazo do mesmo.

passo3) Revisar os prazos d_S em ordem topológica reversa usando a fórmula:

$$d_S = \min (d_S \ , \{ d_{S'} - c_{S'} : S \longrightarrow S' \})$$

onde S e S' são blocos de escalonamento

$c_{S'}$ é o tempo de computação do bloco S'

$d_{S'}$ é o prazo do bloco S'

Usando o exemplo anterior, (Exemplo III), verificar-se-á o funcionamento deste algoritmo. Com os dados da tabela 4.3, as relações de precedência de todos os blocos contidos no intervalo [0,L] são mostradas a seguir :

(Exemplo XIII)

$$c_{11}(1) \longrightarrow c_{22}(1)$$
$$c_{21}(1) \longrightarrow c_{12}(1)$$
$$c_{22}(1) \longrightarrow c_{12}(2)$$
$$c_{11}(1) \longrightarrow c_{12}(1)$$
$$c_{11}(2) \longrightarrow c_{12}(2)$$
$$c_{21}(1) \longrightarrow c_{22}(1)$$
$$c_{12}(1) \longrightarrow c_{11}(2)$$

O grafo é construído a seguir (passo1) e a inicialização dos prazos (passo2) também é calculada :

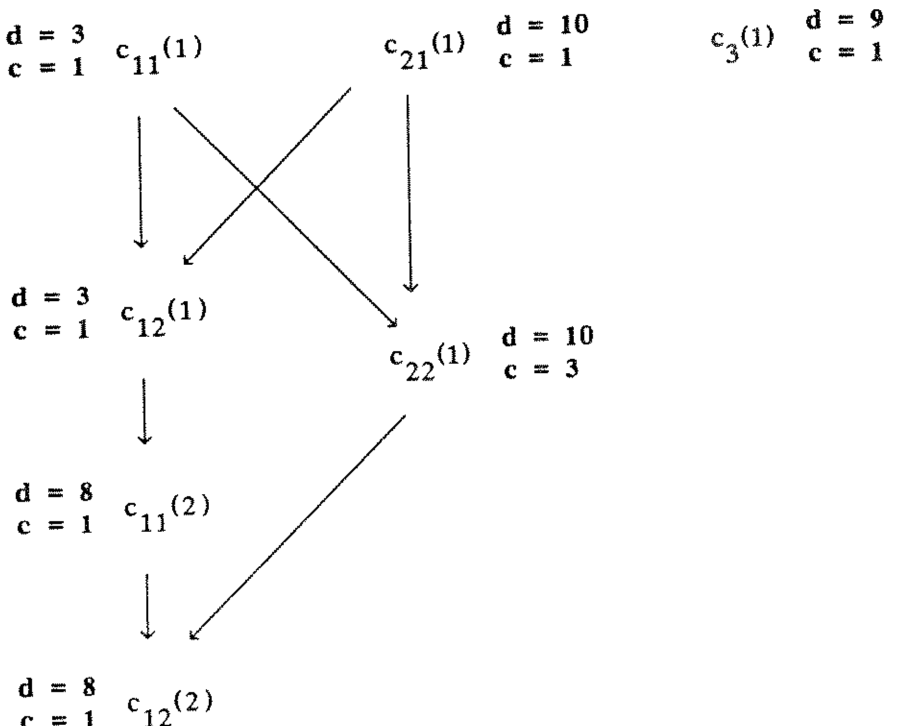


figura 4.8 - Grafo de Relação de Precedência entre os Blocos

No passo3 aplicando-se a fórmula e calculando os prazos :

$$d_S = \min (d_S , \{ d_{S'} - c_{S'} : S \longrightarrow S' \})$$

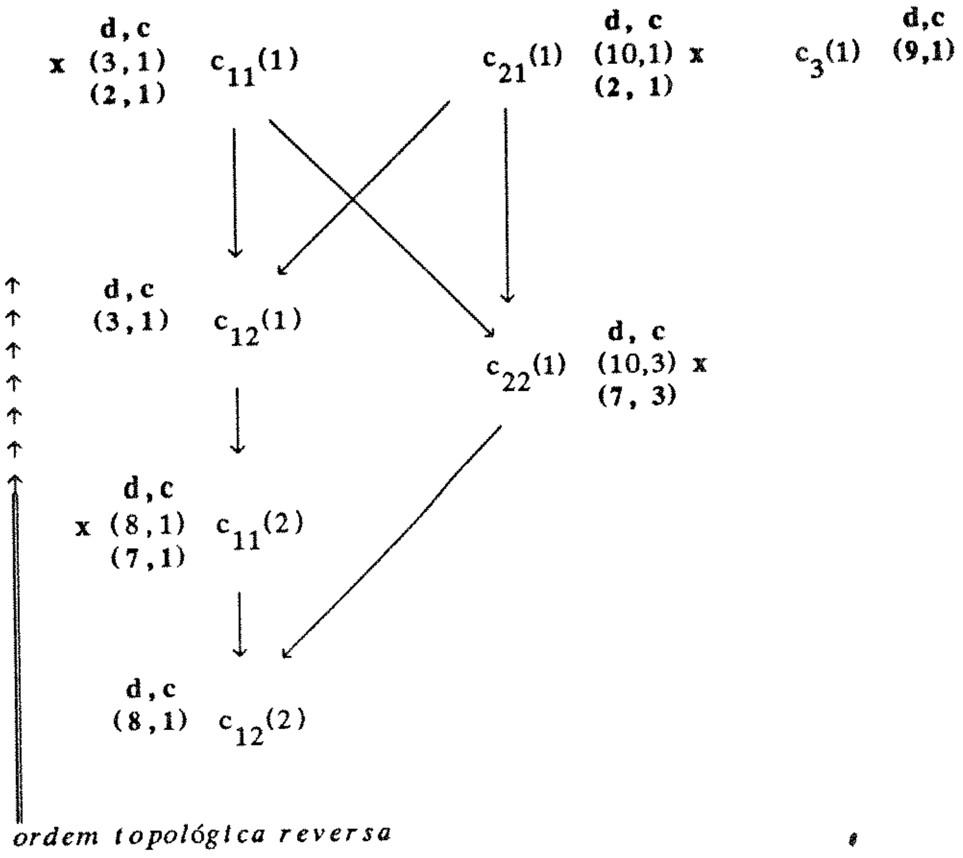


figura 4.9 - Aplicação da Técnica "Prazos Revisados" no Exemplo XIII

Assim, os prazos dos blocos de escalonamento depois de revisados pelo algoritmo são listados abaixo :

$c_{11}(1)$: $d_{11} = 2$
$c_{12}(1)$: $d_{12} = 3$
$c_{11}(2)$: $d_{11} = 7$
$c_{12}(2)$: $d_{12} = 8$
$c_{21}(1)$: $d_{21} = 2$
$c_{22}(1)$: $d_{22} = 7$
$c_3(1)$: $d_3 = 9$

A situação final após as execuções dos processos usando o escalonamento com os novos prazos ajustados é mostrada na figura 4.10.

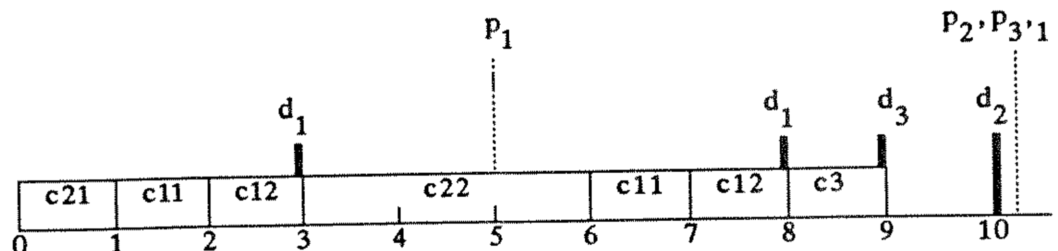


figura 4.10 - Resultado de Escalonamento do Exemplo XIII

Agora, nenhum processo violou sua restrição de temporização; ao contrário, todos terminaram suas execuções antes do vencimento dos seus prazos. No entanto, se o processo T_3 , que não se comunica com nenhum outro processo, possuisse um prazo mais adiantado como por exemplo, igual a 7, então o bloco $c_{12}(2)$, com o seu prazo igual 8, seria executado depois de c_3 o que implica um estouro do prazo para o processo T_1 na segunda instância.

O fato é que o problema continua existindo. Se excluirmos os processos que não possuem comunicação com nenhum outro, deixando apenas processos comunicantes no sistema, será que a técnica de "prazos revisados" funcionaria sem problemas?

4.2.1.2.2. ANÁLISES MAIS PROFUNDAS

A questão citada acima não foi levantada em nenhuma literatura da referência. Assim, para a continuidade deste trabalho iniciaram-se várias análises consideradas importantes para se obter conclusões mais concretas e seguras a respeito da escalonabilidade do algoritmo " Prazos Revisados", como por exemplo, o grau de influência sobre a viabilidade do escalonamento quando os processos são todos comunicantes entre si ou em relação à localização dos pontos de comunicação entre os processos, etc. Enfim, uma solução simples mas indispensável para assegurar a praticabilidade do escalonador será proposta.

Para o melhor entendimento e localização das questões críticas serão vistos mais alguns exemplos.

Toma-se ainda o exemplo mostrado anteriormente (**Exemplo XIII**) com algumas modificações de maneira que o processo T_3 comunique-se também com T_2; assim, os três processos podem ser ditos da mesma classe de equivalência. A nova configuração

das relações interprocessos é apresentada a seguir :
(Exemplo XIV)

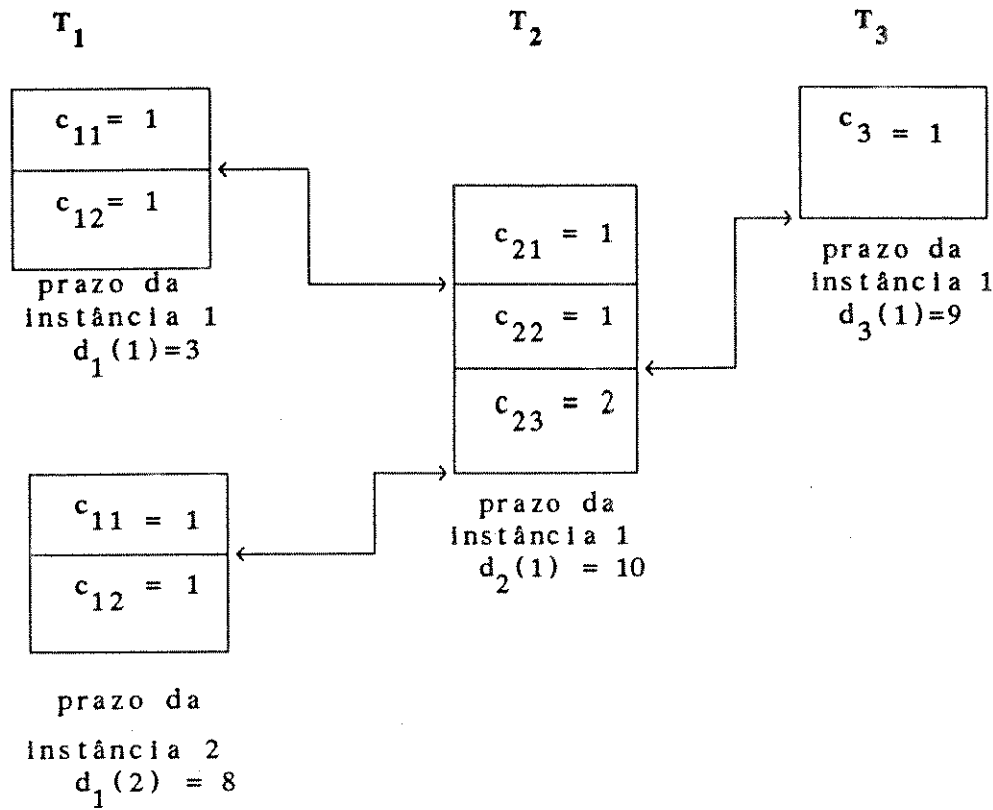


figura 4.11 - Rendezvous entre os Processos do Exemplo XIV

Na figura 4.11 os dois processos T_1 e T_2 comunicam-se nos mesmos pontos que no exemplo anterior, e o processo T_3 comunica-se com T_2, após a execução do bloco c_{22} em T_2 e c_3 em T_3 . Observa-se que o processo T_2 está agora dividido em três blocos de escalonamento. As novas relações de precedência são

$c_{11}(1) \longrightarrow c_{22}(1)$
$c_{21}(1) \longrightarrow c_{12}(1)$
$c_{23}(1) \longrightarrow c_{12}(2)$
$c_{3}(1) \longrightarrow c_{23}(1)$
$c_{11}(1) \longrightarrow c_{12}(1)$
$c_{11}(2) \longrightarrow c_{12}(2)$
$c_{21}(1) \longrightarrow c_{22}(1)$
$c_{22}(1) \longrightarrow c_{23}(1)$
$c_{12}(1) \longrightarrow c_{11}(2)$

Seguindo o algoritmo monta-se o grafo desejado e recalculam-se os prazos com a fórmula :

$$d_S = \min (d_S , \{ d_{S'} - c_{S'} : S \longrightarrow S' \})$$

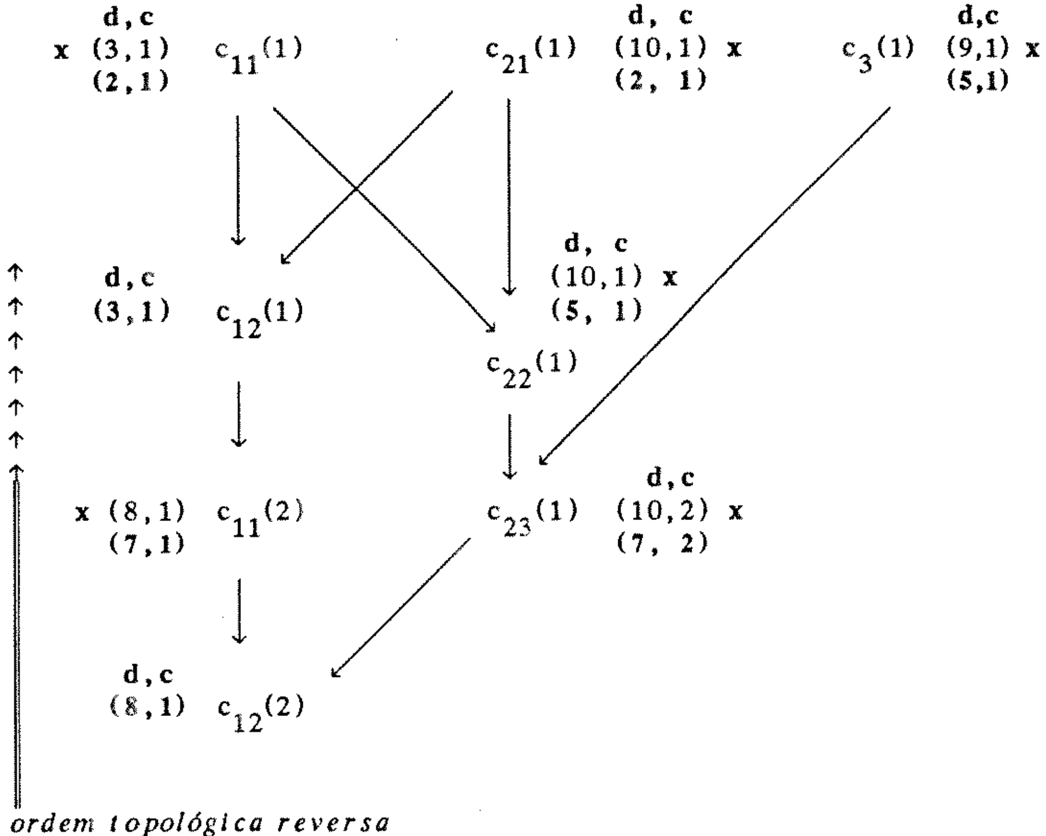


figura 4.12 - Aplicação da Técnica "Prazos Revisados" no Exemplo XIV

O resultado das execuções está na figura 4.13 :

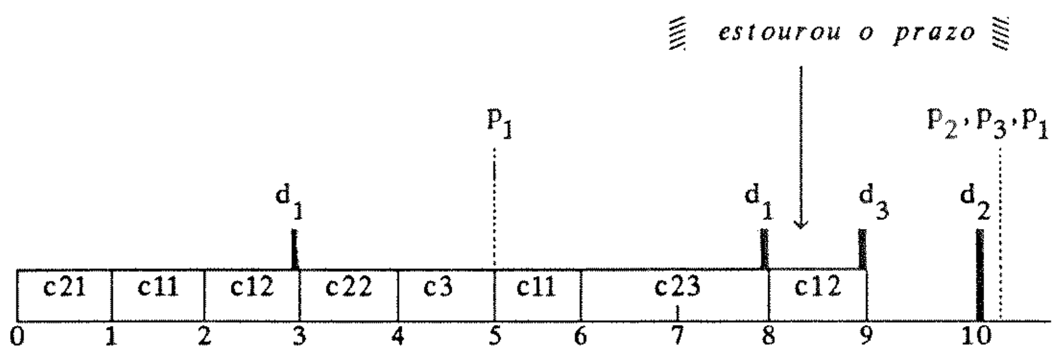


figura 4.13 - Resultado de Escalonamento do Exemplo XIV

Observa -se que a suposição anterior de que "a técnica funciona para um conjunto de processos exclusivamente dependentes, ou melhor, processos que são comunicantes entre si" é refutada pela falha de escalonamento dos processos no exemplo acima.

Continuando a análise, se trocarmos os pontos de comunicação, tem-se :

(Exemplo XV)

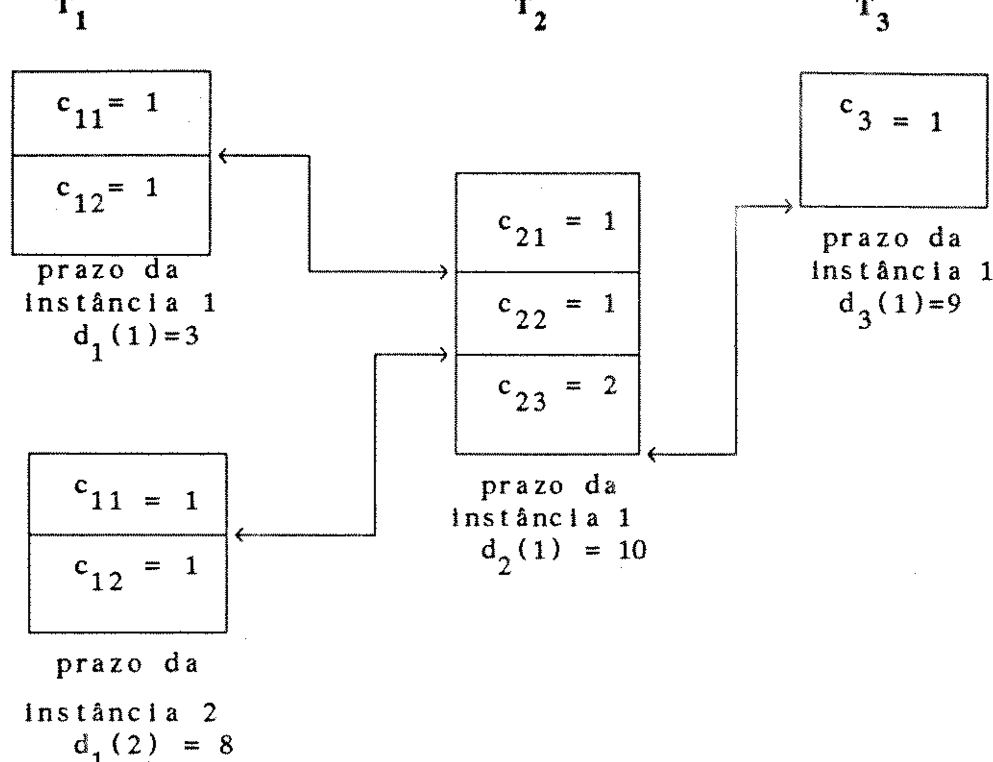


figura 4.14 - Rendezvous entre Processos do Exemplo XV

As relações de precedência são :

$c_{11}(1) \longrightarrow c_{22}(1)$
$c_{21}(1) \longrightarrow c_{12}(1)$
$c_{22}(1) \longrightarrow c_{12}(2)$
$c_{11}(2) \longrightarrow c_{23}(1)$
$c_{11}(1) \longrightarrow c_{12}(1)$
$c_{11}(2) \longrightarrow c_{12}(2)$
$c_{21}(1) \longrightarrow c_{22}(1)$
$c_{22}(1) \longrightarrow c_{23}(1)$
$c_{12}(1) \longrightarrow c_{11}(2)$

Seguindo o algoritmo monta-se o grafo (figura 4.15) desejado e recalculam-se os prazos com a fórmula :

$$d_S = \min (d_S , \{ d_{S'} - c_{S'} : S \longrightarrow S' \})$$

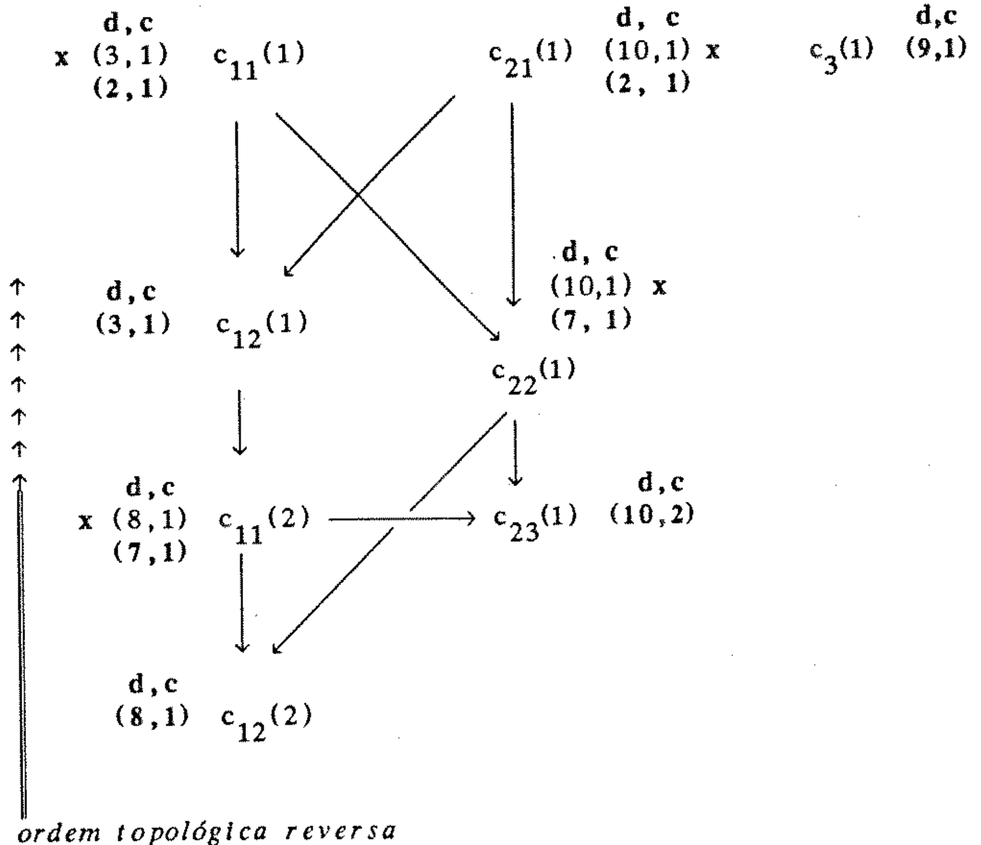


figura 4.15 - Aplicação da Técnica "Prazos Revisados" no Exemplo XV

Portanto, mostra-se o resultado na figura 4.16:

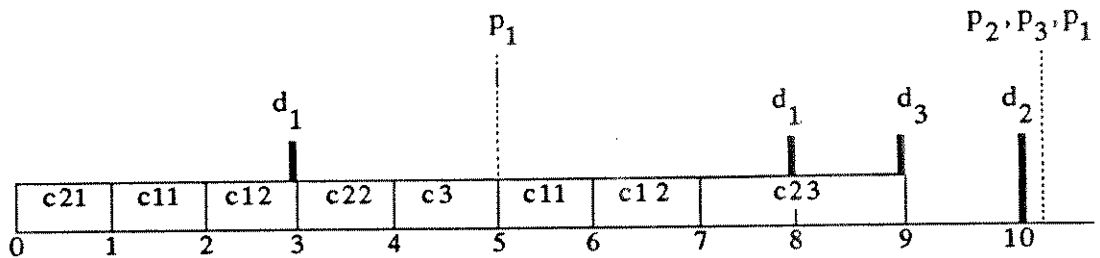


figura 4.16 - Resultado de Escalonamento do Exemplo XV

Observa-se que no instante 4, o bloco $c_{11}(2)$ ainda não estava pronto para a execução, por isso o bloco c_3 foi escalonado naquele momento, mesmo tendo o prazo, $d_3=9$, maior que o do $c_{11}(2)$, $d_{11}=7$.

A ordem do escalonamento está coerente, pois nenhum processo perdeu seu prazo durante execução.

Analisando todos esses exemplos, pode-se concluir que o fato dos processos serem da mesma classe de equivalência não influencia em nada o sucesso ou a falha da técnica de "prazos revisados"; ou seja, independentemente do fato de todos os processos comunicarem-se entre si ou não, o algoritmo não assegura suas execuções antes do vencimento do prazo.

A chave do problema está na estratégia de escolha do lugar em que a comunicação deve ocorrer e, consequentemente, também na questão do tamanho do bloco que deve preceder outros. No (Exemplo V), se o segundo ponto de comunicação entre T_1 e T_2 estivesse marcado um pouco antes, suponha após uma unidade de execução do c_{23} ao invés de duas, então teríamos o tempo suficiente para executar c_{12} antes do vencimento do prazo de T_1. Essa situação é mostrada na figura 4.17

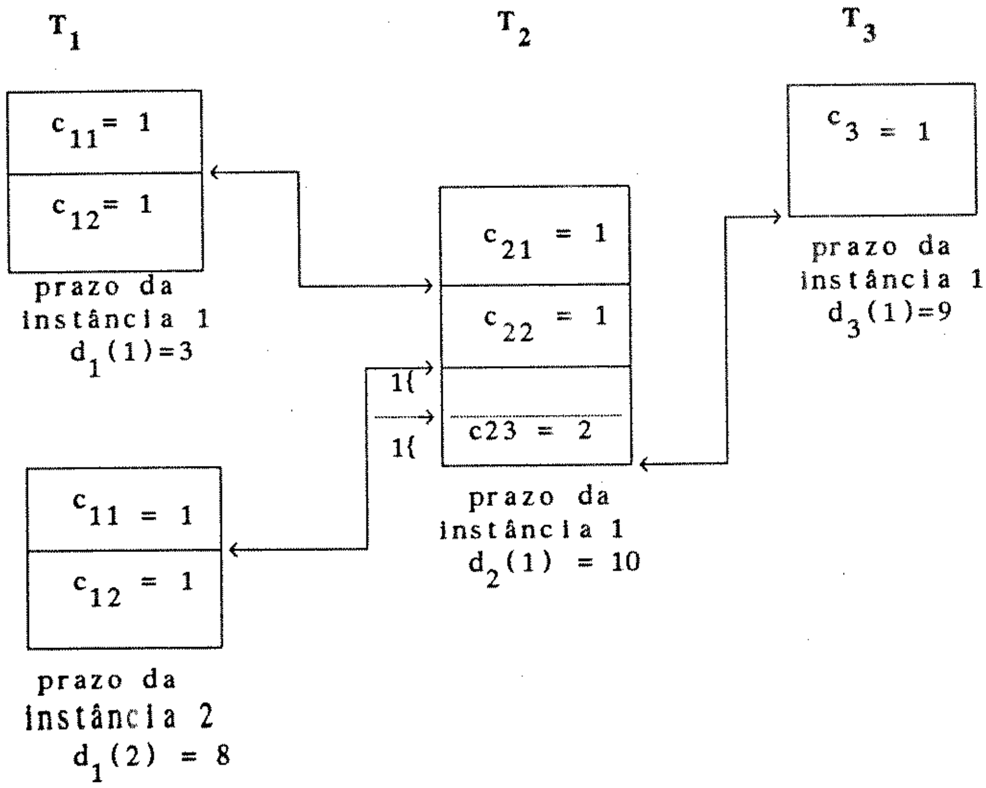


figura 4.17 - Mudança de Ponto de Comunicação do Exemplo XV

Todavia, esta questão levantada não é trivial na análise de solução para os problemas de praticabilidade dos processos no sistema.

4.2.1.2.3. Proposta para Garantir a Viabilidade do Escalonador

Já que o ajuste dos pontos de comunicação para a garantia da viabilidade do escalonador é muito complicado, então pode-se tentar manipular o problema em outras direções, descobrindo-se uma contradição nos prazos já revisados dos processos em relação aos seus tempos de execução. Uma proposta para contornar esses problemas resultantes da aplicação da técnica de "prazos revisados" é efetuar uma verificação final de todos os prazos, após a revisão dos mesmos, de modo que a soma dos tempos de execução dos blocos que tem prazos menores ou iguais ao de um certo bloco B não seja maior que o prazo de B. Uma vez que o prazo de um bloco está determinado, deve haver um espaço de tempo suficiente para que todos os blocos, que têm sua execução antes deste, possam efetivamente ser executados.

Esta técnica pode ser aplicada tanto para um conjunto de processos dependentes, consistindo num único grafo de revisão dos prazos (vide **Exemplo XV**), quanto para um conjunto misto de processos dependentes e independentes, consistindo num grafo para os processos dependentes entre si e os nós "soltos" para os independentes (vide **Exemplo XIII**). Observe-se que o grafo de revisão de prazos resulta num grafo de execução após a revisão, ou seja, uma ordem de execução dentro do grafo. Portanto, é coerente inserir, depois da revisão, os nós de processos independentes dentro do grafo dos dependentes, de acordo com o valor de seu prazo. Obtém-se assim um único grafo de execução englobando todos processos no sistema. Por exemplo, o grafo do **(Exemplo IV)** - figura 4.9 - fica :

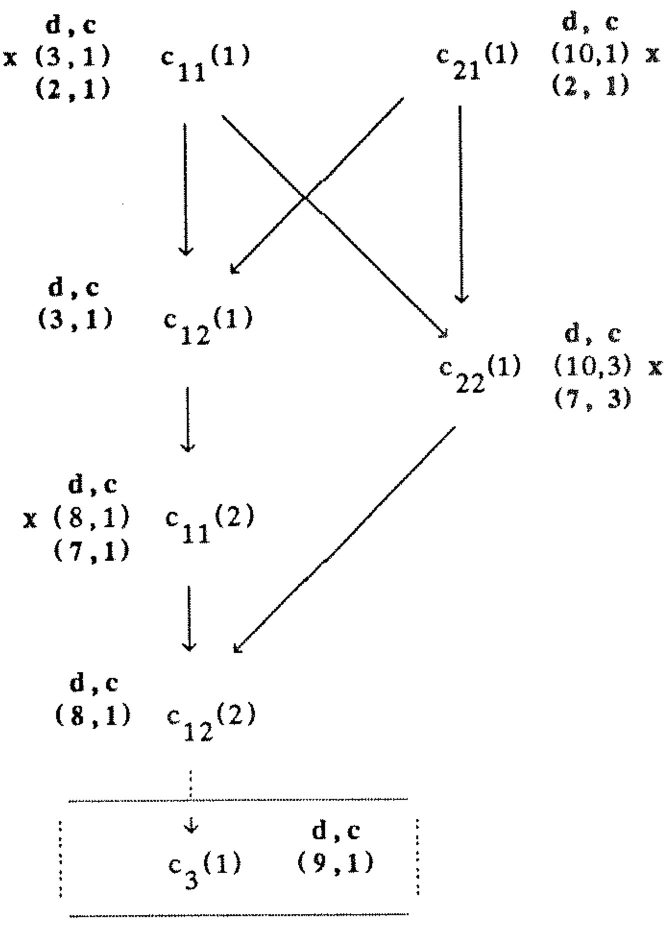


figura 4.18 - Inclusão de Processsos Independentes no Grafo de Execução

Logo, a técnica é usada para a verificação final.

PROPOSTA

Após realizar a revisão dos prazos de todos os processos, toma-se o grafo resultante e realiza-se o seguinte teste :

passo1) Para todos os nós do grafo (começando pelo nó raíz até os nós terminais) faça

passo2) Calcule **soma** onde

soma = soma total dos tempos de execução de todos os blocos-nós antecessores e o do próprio nó

passo3) Se soma for menor ou igual ao prazo do nó referido então pegar o próximo nó e prosseguir o teste, voltando para o passo1). Caso contrário vai para o passo seguinte.

passo4) A viabilidade do escalonamento não é garantida. Falha. Fim.

Por exemplo, na figura 4.12, repare-se que os blocos $c_{11}(2)$ e $c_{23}(1)$ possuem prazos iguais a 7, isto quer dizer que os blocos $c_{11}(1)$, $c_{21}(1)$, $c_{3}(1)$, $c_{12}(1)$, $c_{22}(1)$, $c_{11}(2)$ e $c_{23}(1)$ devem ter suas execuções efetuadas dentro do intervalo [0,7]. Porém, a soma $c_{11}(1) + c_{21}(1) + c_{3}(1) + c_{12}(1) + c_{22}(1) + c_{11}(2) + c_{23}(1) = 8$, o que mostra a impossibilidade de execução destes blocos dentro do prazo determinado. Assim, se este teste complementar for usado depois da aplicação da técnica de "Prazos revisados", o escalonamento correto dos blocos pode ser garantido se o teste for bem sucedido.

***** PROGRAMA REVISA.C *****

Para esse fim contruiu-se um programa em linguagem C denominado REVISA.C o qual calcula os prazos de cada bloco de escalonamento segundo a técnica de "Prazos Revisados"; depois o próprio programa verifica, para cada prazo revisado, se o mesmo tolera as execuções dos blocos que têm prazos menores ou igual ao dele, segundo a proposta descrita acima.

Os parâmetros de entrada deste programa são as relações de precedência entre um par de processos-bloco indicados no grafo montado, e o prazo de execução de cada bloco.

A saída do programa fornece ao usuário não apenas os prazos revisados e verificados dos blocos, mas também, um aviso indicando a praticabilidade do escalonador para esse conjunto de blocos. Se o sinal gerado pelo programa for positivo, então os prazos devem espelhar a viabilidade do escalonamento; caso contrário, deve ocorrer o estouro do prazo de alguns blocos.

4.2.1.3. TESTES E IMPLEMENTAÇÃO

As relações de precedência já são consideradas durante a revisão de prazos feita sobre o grafo. Assim, com a eliminação da dependência interprocessos e a atribuição de um prazo adequado para cada bloco, através da aplicação da técnica "Prazos Revisados", os blocos podem ser vistos como processos independentes um do outro durante a implementação. Para testar esta idéia, utiliza-se o simulador SSE.

Simulador SSE

Todos os blocos de todos os processos dentro do intervalo [0,L] (lembrando que L é o mínimo múltiplo comum dos períodos de todos processos) são agora processos independentes com períodos iguais a L e com os prazos revisados. Porém, é muito comum que os processos com períodos menores ocorram mais que uma vez dentro do intervalo [0,L]. Neste caso, uma condição de "start time" para as ocorrências dos processos a partir da segunda vez deve ser imposta durante a simulação. Nos Exemplos XIII, XIV e XV os blocos $T_{11}(2)$ e $T_{12}(2)$ são da segunda instância do processo T_1; logo, o "start time" deles é 5, ou melhor, eles só podem começar a execução a partir do instante 5. Todos os blocos de primeira instância recebem "start time" 0.

>> Exemplo do teste <<

Tomam-se os dados da tabela 4.3 do (Exemplo XIII) para se mostrar o procedimento de simulação em SSE.

Como já foi visto antes, os prazos do sistema são divididos em blocos nos pontos de comunicação e entre os blocos existem as relações de precedência. Organizando os dados tem-se :

dados sobre o primeiro bloco		Relação de precedência entre dois blocos	dados sobre o segundo bloco	
tempo de execução	prazo		tempo de execução	prazo
1	3	$c_{11}(1) \longrightarrow c_{22}(1)$	3	10
1	10	$c_{21}(1) \longrightarrow c_{12}(1)$	1	3
3	10	$c_{22}(1) \longrightarrow c_{12}(2)$	1	8
1	3	$c_{11}(1) \longrightarrow c_{12}(1)$	1	3
1	8	$c_{11}(2) \longrightarrow c_{12}(2)$	1	8
1	10	$c_{21}(1) \longrightarrow c_{22}(1)$	3	10
1	3	$c_{12}(1) \longrightarrow c_{11}(2)$	1	8

tabela 4.4 - Dados de Entrada para o Programa REVISA.C

Inicialmente, usa-se o programa REVISA para realizar a revisão dos prazos. A forma de entrar os dados segue as relações de precedência, ou seja, o programa solicita primeiramente o nome do bloco que precede qualquer outro bloco e seus tempos de execução e seu prazo; logo em seguida o nome do bloco precedido e também seus dados. Caso os dados de um bloco já tenham sido entrados anteriormente, o programa não repetirá a requisição deles. Assim, a tela de entrada é :

```
<C:\USR\DMOSI\FONTE> revisa

Entre as relacoes de precedencia dos blocos

** Para finalizar digite '00' no nome do bloco


  Nome do bloco : c11(1)
  Deadline / Tempo de exec. --> 3 1
  Precede o bloco : c22(1)
  Deadline / Tempo de exec. --> 10 3

  Nome do bloco : c21(1)
  Deadline / Tempo de exec. --> 10 1
  Precede o bloco : c12(1)
  Deadline / Tempo de exec. --> 3 1

  Nome do bloco : c22(1)
  Precede o bloco : c12(2)
  Deadline / Tempo de exec. --> 8 1

  Nome do bloco : c11(1)
  Precede o bloco : c12(1)


  Nome do bloco : c11(2)
  Deadline / Tempo de exec. --> 8 1
  Precede o bloco : c12(2)

  Nome do bloco : c21(1)
  Precede o bloco : c22(1)


  Nome do bloco : c12(1)
  Precede o bloco : c11(2)

  Nome do bloco : c21(1)
  Precede o bloco : c22(1)

  Nome do bloco : c12(1)
  Precede o bloco : c11(2)

  Nome do bloco : 00
```

Depois de efetuar a entrada de dados o programa mostra ao usuário os prazos já revisados e ainda emite uma mensagem sobre o escalonamento desse conjunto de processos, se praticável ou não.

```
Deadlines resultantes apos a revisao

O blobo c21(1) tem deadline 2
O blobo c11(1) tem deadline 2
O blobo c12(1) tem deadline 3
O blobo c11(2) tem deadline 7
O blobo c22(1) tem deadline 7
O blobo c12(2) tem deadline 8

 !! O escalonamento deste conjunto de processos E' praticavel !!


<C:\USR\DMOSI\FONTE>
```

Comparando os resultados gerados pelo programa com os da figura 4.9 nota-se que são idênticos.

Como a saída do programa indica a viabilidade do escalonamento desse conjunto de dados, essa afirmação pode ser confirmada aplicando-se o SSE.

Considere cada bloco c_{ij} dentro do intervalo [0,L] como sendo um processo independente, definido como **processo-bloco** na simulação, com seu novo prazo revisado, d_{ij}, e o período p_{ij} igual a L.

Porém, um outro parâmetro importante neste caso é o "Start Time", principalmente para aqueles processos-bloco da segunda instância em diante, pois estes, pela natureza original do processo, devem começar a partir do início da segunda instância

Antes de determinar uma regra para o cálculo do "Start Time" dos processos-bloco, uma observação importante deve ser citada. Normalmente, um processo periódico que impõe uma condição de "Start Time" diferente de 0, ou seja, ele deve ficar pronto depois de certo tempo do começo de cada período, pode ser interpretado como um processo com "Start Time" igual a 0 deslocando a sua ativação inicial em um intervalo correspondente ao "Start Time" original. Assim, o processo será repetido, após essa transformação, com uma defasagem de tempo igual a seu "Start Time" original (figura 4.19). Portanto, essa observação leva a uma suposição inerente, em geral, onde os processos periódicos são ditos possuir "Start Time" igual a 0. Esta transformação

será usada após o cálculo de "Start Time" dos processos-bloco para simplificar a simulação.

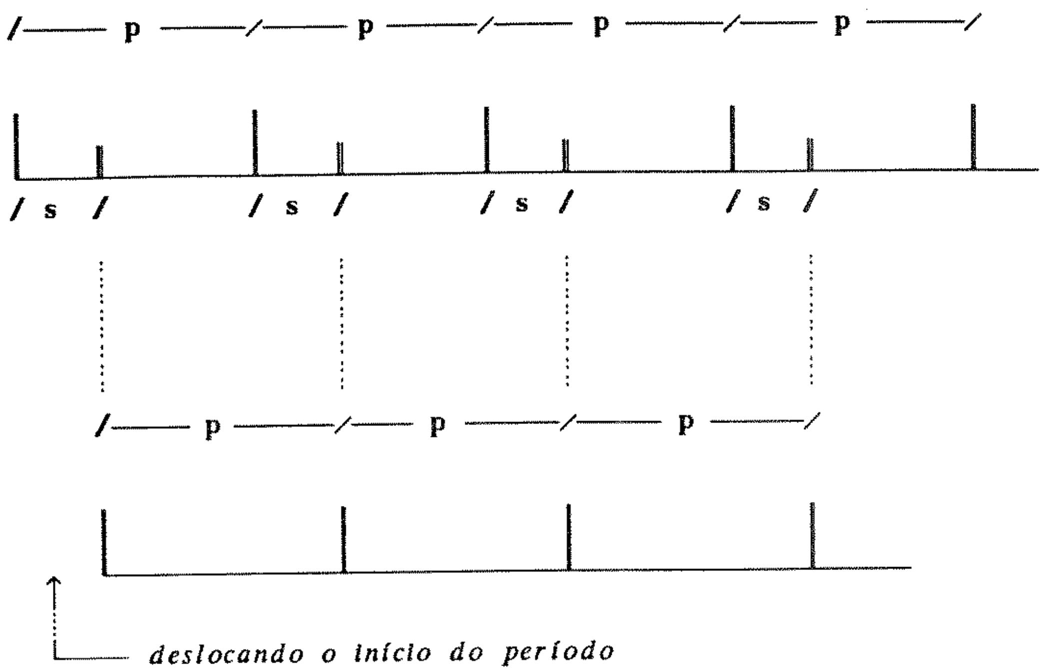


figura 4.19 - Eliminação de "Start Time" de um Processo Periódico

O cálculo de "Start Time", $S_{ij}(k)$, de um processo-bloco $c_{ij}(k)$ pode ser formulado como :

$$S_{ij}(k) = ((k - 1) * p_i) + S_i$$

onde $S_{ij}(k)$ é o "Start Time" do j-ésimo bloco do processo T_i na k-ésima instância

k é o número de instância

p_i é o período do processo T_i

S_i é o "Start Time" do processo T_i que normalmente é 0

bloco	prazo	temp_exec	Start Time	período	período na simulação	id. na simulação
$c_{11}(1)$	2	1	0	5	10	P1
$c_{12}(1)$	3	1	0	5	10	P2
$c_{11}(2)$	7	1	5	5	10	P6
$c_{12}(2)$	8	1	5	5	10	P7
$c_{21}(1)$	2	1	0	10	10	P3
$c_{22}(1)$	7	3	0	10	10	P4
$c_{3}(1)$	9	1	0	10	10	P5

tabela 4.5 - Dados de Entrada para o Simulador SSE

Deve ser observado que existem, neste caso, dois processos-bloco $c_{11}(2)$ e $c_{12}(2)$ que possuem "Start Time" diferente de 0; os demais possuem "Start Time" igual a 0. Assim, para facilitar a simulação no SSE, basta aplicar a idéia citada anteriormente nos processos-bloco com "Start Time" diferente de 0, deslocando seus tempos de início para o "Start Time".

Então o modo de simular estes processos-bloco consiste em começar os da primeira instância no instante 0 e inserir os da segunda instância em diante durante o funcionamento do sistema de acordo com seu "Start Time". Assim, com os dados da tabela 4.5 os blocos $c_{11}(1)$, , $c_{12}(1)$, $c_{21}(1)$, $c_{22}(1)$ e $c_{3}(1)$ (que são P1, P2, P3, P4 e P5, respectivamente, durante a simulação) ficam prontos no início e os blocos $c_{11}(2)$ e $c_{12}(2)$ (P6 e P7 na simulação) serão inseridos no instante 5, depois do sistema começar.

4.2.1.4. Lemas e Teoremas

Lema 1

Suponha um conjunto de processos periódicos que trocam mensagens através do mecanismo de rendezvous. A praticabilidade deste conjunto não é afetada pela atualização dinâmica dos prazos (técnica citada anteriormente); além disso,

sempre que o prazo dinâmico de um processo T_i é mais próximo que o do processo T_j, então o ato de escalonar T_i antes de T_j não viola qualquer restrição de precedência envolvida nos dois processos.

Observação

Sejam S e S' dois blocos de escalonamento em [0,L]. Aplicada a técnica de prazos revisados tem-se a fórmula da revisão : $d_s < d_{s'}$ se $S \longrightarrow S'$. Assim, num dado momento do sistema, se o bloco atual de T_i possui prazo mais próximo que o bloco de T_j, a execução do bloco de T_i é realizada antes da execução do bloco de T_j, respeitando a relação de precedência interprocessos.

Porém, não se pode afirmar nada em relação à técnica de prazos revisados para um escalonamento não praticável de um conjunto de processos periódicos pois, segundo o lema, a aplicação da técnica não afeta a praticabilidade de um escalonamento já praticável, mas não garante a possibilidade de tornar praticável um escalonamento não praticável.

Teorema 1

Se existe um escalonador praticável para um conjunto de processos que possui restrição de comunicação do tipo rendezvous, então estes podem ser escalonados pelo algoritmo "earliest deadline" modificado (aplicando a técnica referida) como apresentado anteriormente.

Observação

Como o lema anterior garante a não violação das relações de precedência imposta pelas primitivas de rendezvous dos processos com os prazos revisados, então o escalonador "Earliest Deadline", que escalona sempre aquele processo de prazo mais próximo, pode ser usado sem causar nenhum problema.

4.2.2. MONITOR "KERNELIZED" (AMB 6)

4.2.2.1. DEFINIÇÃO DO PROBLEMA

Exclusão mútua é um dos problemas mais importantes na programação concorrente e nos sistemas de tempo real. A partir dela podemos abstrair muitos dos problemas de sincronização entre processos.

Dois processos T_i e T_j excluem-se mutuamente se a execução de alguma atividade de um deles não se sobrepõe à do outro processo; em outras palavras, se essas duas atividades tentam executar simultaneamente, então a CPU é obrigada a escolher somente uma delas para a execução, e a outra deve ser bloqueada. Normalmente, essa situação é encontrada no caso da alocação de recursos comuns para um grupo de processos.

Assim, a região crítica e seus protocolos de entrada geralmente são usados para modelar o problema de exclusão mútua. Porém, a repetição desnecessária do corpo de cada região crítica em todos os processos relacionados pode aumentar a probabilidade de erros de código na programação. Neste caso, o monitor pode ser usado para evitar tais problemas. O monitor pode ser implementado como um processo especial, o qual realiza alguns serviços a pedido de outros processos ordinários. Toma-se a versão simplificada do conceito de monitor de Hoare [HOA 74] e tem-se a seguinte sintaxe :

```
Processo  < nome_do_monitor >
     rendezvous  (qualquer processo T_i)
     {  corpo do serviço  }
     rendezvous  (T_i)
end  < nome_do_monitor >
```

Um processo ordinário solicita os serviços do monitor, fazendo rendezvous com o mesmo; se dois ou mais processos estiverem pedindo serviço ao mesmo tempo, o escalonador pode escolher aleatoriamente um único dentre eles para ser atendido e realizar o rendezvous correspondente com o monitor. Depois que o corpo do serviço foi executado, o monitor tenta outro rendezvous com o processo para o qual foi

prestado o serviço. Apesar do monitor não ter nenhuma restrição de temporização explícita, ele não deve atrasar o prazo atual do processo para o qual presta o seu serviço. Por outro lado, o processo que requer o serviço deve conter duas primitivas de rendezvous destinadas ao monitor. Isto produz o mesmo efeito que inserir a macro do corpo de serviço dentro do processo, tendo a garantia de que nenhum outro processo pode executar o código do monitor ao mesmo tempo. A característica de exclusão mútua faz lembrar que o monitor, neste caso, é tratado como uma região crítica.

4.2.2.2. Modelagem do Monitor

Será definida uma condição para contornar o problema de exclusão mútua. Suponhamos que o núcleo de tempo real aloca, de cada vez, um tempo de processamento não interrompível para o processo. Este tempo é chamado quantum **q** e seu tamanho é definido de acordo com o maior monitor do sistema. A intenção de proteger a execução dentro de uma região crítica sem interrupções por outros processos é coerente. Assim, **q** é uma unidade básica de alocação de tempo do núcleo. Divide-se então cada bloco de escalonamento, definido anteriormente, em um número inteiro de quantuns, ou seja, o tamanho do bloco é múltiplo exato de **q**. Como um processo é constituído por vários blocos de escalonamento, este também será composto por um número inteiro de quanta, os quais podem ser encarados como miniblocos de escalonamento.

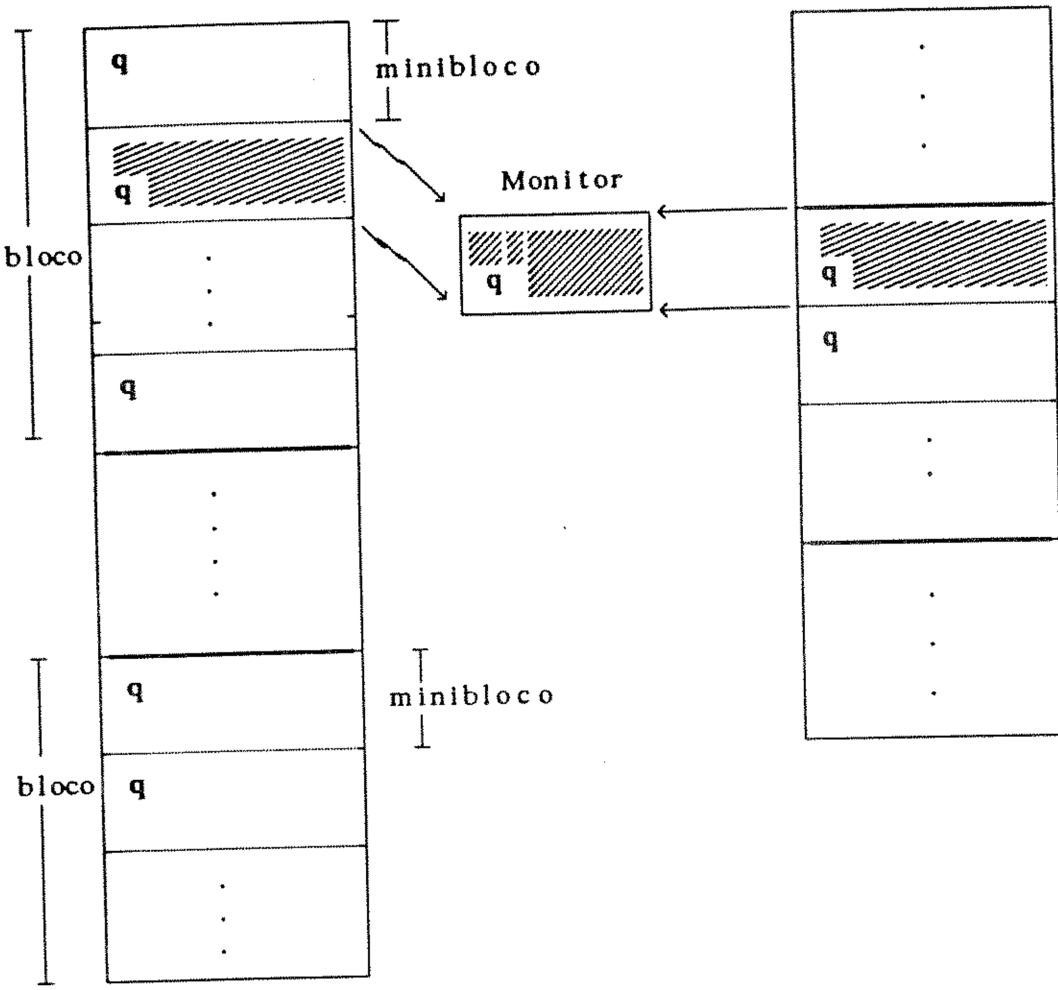


figura 4.20 - Relacionamento entre Processo, Blocos e Miniblocos

Entre os blocos de escalonamento devem ocorrer comunicações de rendezvous como foi analisado no item anterior.

Sendo o tamanho do quantum q determinado de acordo com o tamanho dos monitores do sistema, então quanto menores forem as regiões críticas, melhor será o desempenho do escalonador. Com esse critério de alocação do tempo de processamento não-interrompível, as regiões críticas não mais impõem restrições ao escalonador.

Embora a estrutura de quantuns resolva o problema de exclusão mútua, as restrições de temporização dos processos ainda não são tratadas com relevância pelo escalonador "earliest deadline". Um exemplo que mostra essa falha é :

(Exemplo XVI)

Sejam T_1 e T_2 dois processos periódicos. Existe uma região crítica compartilhada por c_1 e c_{22}, ou seja, eles possuem a mesma região crítica em c_1 e c_{22}, respectivamente. Como c_1=c_{22}= 2 então ao quantum será atribuído o valor 2. A seguir são mostrados os dados (ou parâmetros) dos dois processos.

	T_1	T_2
c_i	$c_1 = 2$	$c_{21} = 2$ $c_{22} = 2$
p_i	5	10
d_i	2	10

tabela 4.6 - Dados dos Processos do Exemplo XVI

O processo T_2 tem seu tempo de execução total igual a 4 o que satisfaz a condição de que o tamanho do processo seja múltiplo exato de q pois 4 = 2 * q.

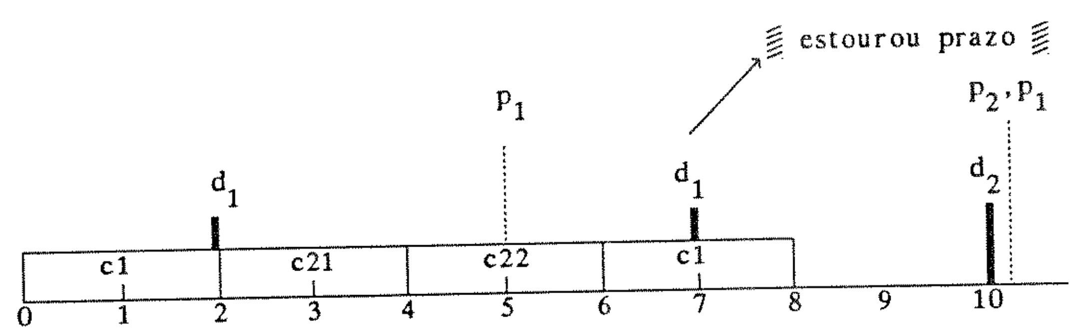


figura 4.21 - Resultado de Escalonamento do Exemplo XVI

A segunda instância de T_1 perdeu seu prazo no instante 7. Observa-se que o escalonamento de $c_{22}(1)$ no instante 4 atrasou a execução do minibloco $c_1(2)$,

o qual já estava pronto desde o instante 5 mas não podia interromper a execução do quantum de $c_{22}(1)$; consequentemente estourou seu prazo antes de finalizar sua execução. A decisão mais inteligente é não escalonar o minibloco $c_{22}(1)$ no intervalo [4,5], deixando a CPU ociosa,de modo que $c_1(2)$ seja executado assim que estiver pronto, no instante 5.

Logo, adota-se a técnica de "região proibida" para eliminar tal problema. A idéia central desta é encontrar, usando os valores de prazo e de tempo de execução dos processos, um intervalo específico no qual o escalonador não pode alocar um novo quantum de tempo de execução para qualquer minibloco. No exemplo anterior, a região proibida é [3,5] como será mostrado adiante, então o escalonamento do minibloco $c_{22}(1)$ no instante 4 não seria permitido e sim adiado para o instante 7.

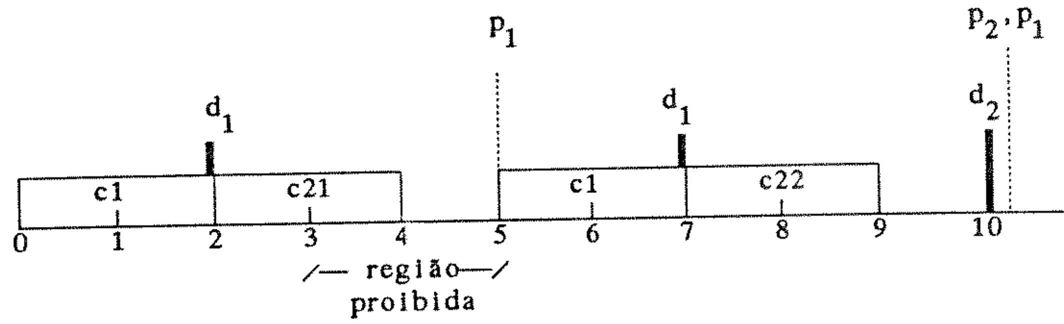


figura 4.22 - Funcionamento da Técnica "Região Proibida"

A figura acima mostra o aspecto viável do escalonamento aplicando o algoritmo da "região proibida".

4.2.2.3. Definição da Técnica de "Região Proibida"

Deverá ser montada uma base de dados a qual conterá um conjunto de regiões proibidas no intervalo [0,L]. Aqui, L é o mínimo múltiplo comum de todos os períodos no sistema, como já foi definido no item anterior (rendezvous). Essa base de dados também é reciclada a cada L unidades de tempo para atualizar as regiões proibidas. Assim, o escalonador earliest deadline alocará um quantum de tempo de execução para um processo pronto no instante t, desde que seja o processo de prazo mais próximo e não bloqueado pelo rendezvous e t não esteja dentro de nenhuma região proibida.

Para computar o conjunto de regiões proibidas, considera-se cada processo contendo uma cadeia de miniblocos de escalonamento, cada um dos quais equivalente a um quantum. Os miniblocos formam uma ordem parcial imposta pela restrição interprocessos ou intraprocessos.

4.2.2.3.1. ALGORITMO DA TÉCNICA

**** ETAPA I ****

passo 1) Organizar em ordem topológica direta os miniblocos de escalonamento gerados em [0,L].

passo 2) Inicializar o tempo de requisição de execução da k-ésima instância de cada minibloco do processo T_i com valor $(k-1)*p_i$

$$r_{i,j}(k) = (k-1)*p_i$$

onde p_i é o período do processo T_i

passo 3) Revisar os tempos de requisição em ordem topológica direta usando a fórmula

$$r_s = \text{MAX} (r_s , \{ r_{s'} + q \; : \; s' \longrightarrow s \})$$

passo 4) Organizar os miniblocos gerados em [0,L] em ordem topológica reversa

passo 5) Inicializar o prazo (deadline) de cada minibloco j do processo T_i da k-ésima instância com valor $(k-1)*p_i+d_i$

$$d_{ij}(k) = (k-1) * p_i + d_i$$

passo 6) Revisar os prazos de todos os miniblocos em ordem topológica reversa usando a fórmula

$$d_s = \text{MIN} (d_s , \{ d_{s'} - q \; : s \longrightarrow s' \})$$

onde d_s é o prazo do minibloco s

q é o quantum

s $\longrightarrow$ s' significa que o minibloco s precede o minibloco s'.

Nessa etapa obteve-se para cada minibloco do sistema o tempo de requisição r_s e o prazo d_s.

****** ETAPA II ******

passo 1) Organizar os tempos de requisição em ordem cronológica reversa. Inicialmente o conjunto de região proibida é vazio.

passo 2) Para cada minibloco T_s do sistema, com seu tempo de requisição r_s e prazo d_s, faça :

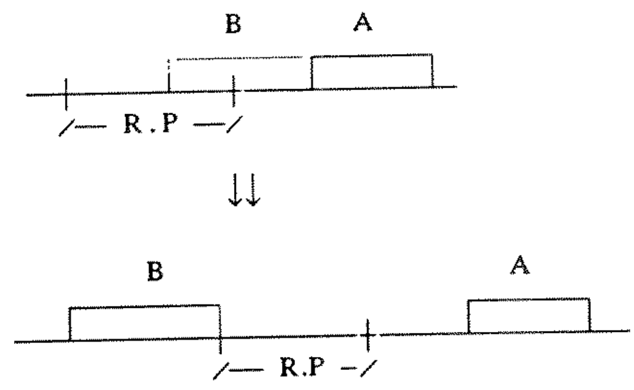
Para cada d tal que $L \geq d \geq d_s$ faça :

Determinar $n_{rs,d}$, o número total de miniblocos escalonados no intervalo $[r_s, d]$, ou seja,

$n_{rs,d}$ = número dos miniblocos $T_{s'}$,

onde $r_{s'} \geq r_s$ e $d_{s'} \leq d$

passo 3) Para todos os intervalos $[r_s, d]$ obtidos no passo anterior, escalonar seus $n_{rs,d}$ miniblocos dentro dos quais destaca-se o tempo mais tardio, $s_{r,d}$, no qual o primeiro deles deve ser escalonado. Uma forma mais fácil de verificar o valor de $s_{r,d}$ é começar de trás para frente pelo instante d e alocar os miniblocos o mais à direita possível, respeitando suas restrições de temporização. Caso essa tentativa de alocação de blocos resulte numa sobreposição dos mesmos dentro de algumas regiões proibidas, então deve-se adiantar a alocação desses blocos para o lado esquerdo da região proibida e prosseguir nas demais alocações.



Depois de determinar $s_{rs,d}$ verifica-se :

Caso $s_{r,d} < r_s$:

Não foi possível ajustar todos miniblocos dentro do intervalo $[r_s, d]$,

FALHA !

Caso $s_{r,d} \geq r_s$:

Se $s_{r,d} < r_s + q$ Então

Inserir o intervalo $(s_{r,d} - q\ ,\ r_s)$ dentro do conjunto das regiões proibidas. Se $(s_{r,d} - q)$ for negativo então toma-se $(0, r_s)$ como região proibida.

Aplicando o algoritmo no **Exemplo XVI** tem-se, passo a passo, o processo para encontrar as regiões proibidas.

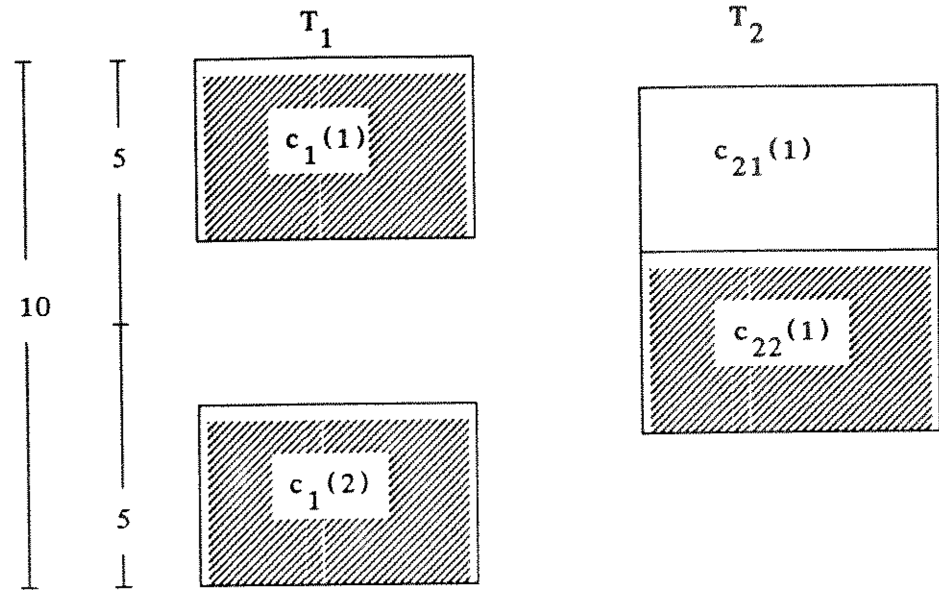


figura 4.23 - Configuração Global dos Processos do Exemplo XVI

Neste caso, L é 10 pois é o mínimo múltiplo comum dos períodos $p_1=5$ e $p_2=10$. Não existe nenhuma restrição de precedência interprocessos, apenas intraprocessos. Por isso as relações de precedência estabelecidas são :

$c_1(1) \longrightarrow c_1(2)$

$c_{21}(1) \longrightarrow c_{22}(1)$

****** ETAPA I ******

passo 1) Construir a ordem topológica direta dos miniblocos :

Obs. Para este sistema temos dois grafos independentes

passo 2) Inicializar o tempo de requisição de cada um :

$$r_{i,j}(k) = (k-1)*p_i$$

passo 3) Usando a fórmula

$$r_s = \text{MAX}\ (\ r_s\ ,\ \{\ r_{s'} + q\ :\ s' \longrightarrow s\ \}\ )$$

revisar os tempos de requisição :

$r_1(1)=0$ $c_1(1)$ → $c_1(2)$ $r_1(2)=5$

$c_{21}(1)$ $r_{21}(1)=0$ → $c_{22}(1)$ $r_{22}(1)=0$ x

$r_{22}(1)=2$

passo 4 e passo 5) Inicializar o prazo de cada minibloco em T_i com $(k-1) * p_i + d_i$, em ordem topológica reversa

passo 6) Os prazos dos miniblocos são revisados com a fórmula :

$$d_s = \text{MIN} (d_s \ , \{ d_{s'} - q \ : s \longrightarrow s' \})$$

$d_{21}(1)=10$ x

$d_1(1)=2$ $c_1(1)$ $c_{21}(1)$ $d_{21}(1)=8$

↓ ↓

$d_1(2)=7$ $c_1(2)$ $c_{22}(1)$ $d_{22}(1)=10$

Depois de calcular os tempos de requisição e os prazos prossegue-se com a etapa II do algoritmo.

**** ETAPA II ****

passo 1) Organizar os miniblocos em ordem cronológica de tempos de requisição como mostrado abaixo :

	$c_1(1)$	$c_{21}(1)$	$c_{22}(1)$	$c_1(2)$
r_i	0	0	2	5
d_i	2	8	10	7

passo 2)

(1) $T_s = c_1(1)$ { $r_s=0$ e $d_s=2$ }

Para d = 2, 8, 10, 7 ($2 \le d \le 10$)

faça

d_i	$n_{rs,d}$ nro total de rs' $r_s, \ge r_s$ e $d_s, \le d$	miniblocos
d = 2	$n_{0,2} = 1$ ($r_s, \ge 0$ e $d_s, \le 2$)	$c_1(1)$
d = 8	$n_{0,8} = 3$ ($r_s, \ge 0$ e $d_s, \le 8$)	$c_1(1)$, $c_{21}(1)$ e $c_1(2)$
d = 10	$n_{0,10} = 4$ ($r_s, \ge 0$ e $d_s, \le 10$)	$c_1(1)$, $c_{21}(1)$ $c_{22}(1)$ e $c_1(2)$
d = 7	$n_{0,7} = 2$ ($r_s, \ge 0$ e $d_s, \le 7$)	$c_1(1)$ e $c_1(2)$

(2) $T_s = c_{21}(1)$ { $r_s=0$ e $d_s=8$ }

Para d = 8, 10 ($8 \le d \le 10$)

faça

d_i	$n_{rs,d}$ nro total de rs' $r_s, \ge r_s$ e $d_s, \le d$	miniblocos
d = 8	$n_{0,8} = 3$ ($r_s, \ge 0$ e $d_s, \le 8$)	$c_1(1)$, $c_{21}(1)$ e $c_1(2)$
d = 10	$n_{0,10} = 4$ ($r_s, \ge 0$ e $d_s, \le 10$)	$c_1(1)$, $c_{21}(1)$ $c_{22}(1)$ e $c_1(2)$

(3) $T_s = c_{22}(1)$ { $r_s=2$ e $d_s=10$ }
Para d = 10 ($10 \le d \le 10$)
faça

d_i	$n_{rs,d}$ nro total de rs' $r_s{'} \ge r_s$ e $d_s{'} \le d$	miniblocos
d = 10	$n_{2,10} = 2$ ($r_s{'} \ge 2$ e $d_s{'} \le 10$)	$c_{22}(1)$ e $c_1(2)$

(4) $T_s = c_1(2)$ { $r_s=5$ e $d_s=7$ }
Para d = 7, 8, 10 ($7 \le d \le 10$)
faça

d_i	$n_{rs,d}$ nro total de rs' $r_s{'} \ge r_s$ e $d_s{'} \le d$	miniblocos
d = 8	$n_{5,8} = 1$ ($r_s{'} \ge 5$ e $d_s{'} \le 8$)	$c_1(2)$
d = 10	$n_{5,10} = 1$ ($r_s{'} \ge 5$ e $d_s{'} \le 10$)	$c_1(2)$
d = 7	$n_{5,7} = 1$ ($r_s{'} \ge 5$ e $d_s{'} \le 7$)	$c_1(2)$

passo 3) Arrumar os dados obtidos no passo anterior e determinar, para $n_{rs,d}$, o valor de $s_{r,d}$ e consequentemente a região proibida.

OBS : a região proibida R.P. é definido como $(s_{r,d} - q \, , \, r_s)$

$n_{rs,d}$	miniblocos escalonados	escalonamento	$s_{r,d}$	$s_{r,d} \geq r_s$	$s_{r,d} < r_s+q$	R.P.
$n_{0,2}=1$	$c_1(1)=2$	$c_1(1)$: 0–2	0	sim	sim	[0,0]
$n_{0,8}=3$	$c_1(1)=2$ $c_{21}(1)=2$ $c_1(2)=2$	$c_1(1)$: 0–2; $c_{21}(1)$: 3–5; $c_1(2)$: 5–7; 8	0	sim	sim	[0,0]
$n_{0,10}=4$	$c_1(1)=2$ $c_{21}(1)=2$ $c_{22}(1)=2$ $c_1(2)=2$	$c_1(1)$: 0–2; $c_{21}(1)$: 3–5; $c_1(2)$: 5–7; $c_{22}(1)$: 8–10	0	sim	sim	[0,0]
$n_{0,7}=2$	$c_1(1)=2$ $c_1(2)=2$	$c_1(1)$: 0–2; $c_1(2)$: 5–7	0	sim	sim	[0,0]
$n_{2,10}=2$	$c_{22}(1)=2$ $c_1(2)=2$	2; $c_1(2)$: 5–7; $c_{22}(1)$: 8–10	5	sim	não	[0,0]
$n_{5,7}=1$	$c_1(2)=2$	$c_1(2)$: 5–7	5	sim	sim	[0,0] [3,5]
$n_{5,8}=1$	$c_1(2)=2$	$c_1(2)$: 5–7	5	sim	sim	[0,0] [3,5]
$n_{5,10}=1$	$c_1(2)=2$	$c_1(2)$: 5–7; 10	5	sim	sim	[0,0] [3,5]

tabela 4.7 - As Regiões Proibidas Calculadas do Exemplo XVI

Esse modelo de monitor "kernelized" impõe uma restrição ao escalonador, pois não é permitida a preempção das regiões críticas por qualsquer outros miniblocos. Contudo, o efeito do atraso devido a esta restrição é melhor tolerado, se as regiões críticas no sistema forem as menores possíveis, pois o tamanho delas influencia diretamente no valor do quantum, unidade básica de alocação de tempo de processamento sem interrupção. Com um quantum maior, o escalonador torna-se menos flexível para alocar um processo (ou minibloco) pronto para execução imediata.

4.2.2.4. Testes e Implementação

Usa-se aqui o núcleo de tempo real DEADMOSI para testar o algoritmo da técnica de Região Proibida que contorna os problemas de exclusão mútua, quando o escalonador empregado é "Earliest Deadline".

Na certa, segundo o algoritmo, um escalonador "Earliest Deadline" deve ser modificado com alguns ajustes de modo que o algoritmo possa funcionar. Porém, muitas vezes as alterações num núcleo pronto e fechado não seriam possíveis; por isso, uma tentativa de elaborar, a nível de usuário, algumas adaptações para integrar os testes foi realizada neste trabalho.

****** Proposta para os Testes ******

Cria-se uma base de dados contendo todas as regiões proibidas pré-calculadas no intervalo $[0,L]$ que por sua vez são atribuídas ao sistema durante a inicialização. Ainda assim, é preciso um processo de controle que assuma o papel de simular a unidade básica de escalonamento, q (quantum do núcleo), onde a verificação de qual processo será escalonado para ser executado é feito.

Na verdade, este processo de controle possui seu período igual ao tamanho do quantum determinado, ou seja, ele é ativado em cada q unidades. Além disso, ele sempre possui a maior prioridade de execução em relação aos processos do sistema; isto é coerente se os tempos de execução de todos os processos forem múltiplos exatos de q, ou seja, o período do processo de controle. A idéia central do processo de controle é : antes de alocar um quantum de tempo de execução para qualquer processo no sistema, ele verifica a relação entre o instante atual com as regiões proibidas e decide se força a CPU a ficar ociosa ou entrega-a para executar outros processos propriamente ditos.

****** Algoritmo do Processo de Controle ******

passo1) Determinar o conjunto de regiões proibidas se for necessário (em cada L unidades de tempo)

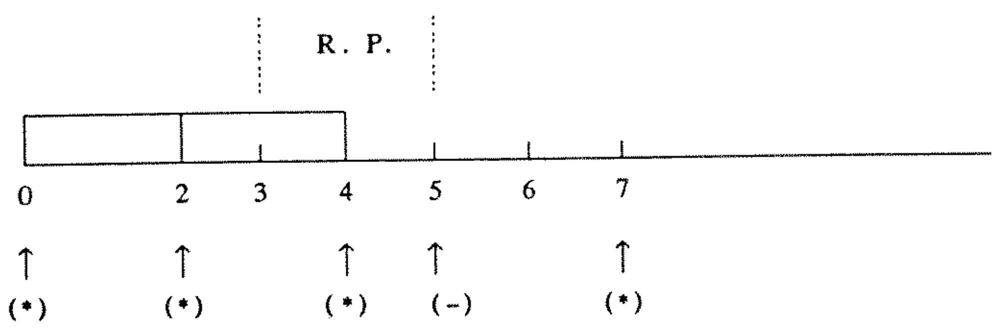
passo2) Se o instante atual t estiver dentro da região proibida então vai para passo3 senão vai para passo4

passo3) O processo de controle não aloca o tempo de execução q para nenhum processo; executa ociosamente até o fim da região proibida referida. Ao terminar a região proibida, o processo de controle renova seu período a partir deste instante.

passo4) Entregar a CPU para o minibloco de maior prioridade, isto é, menor prazo (deadline), no momento. Fim.

O fato do processo de controle ser o mais prioritário no sistema contribui com a função de efetuar as verificações citadas antes da execução dos miniblocos dos processos.

Observa-se que a renovação do período do processo de controle, após o término da região proibida, é importante, pois sem ela pode haver uma confusão no controle da execução do processo referido. Tal situação pode ser melhor explicada através da figura 4.24.



(*) : o processo de controle é ativado para a verificar se o sistema está em R.P.

(-) : o processo reativa seu período ao sair da R.P.

figura 4.24 - Função do Processo de Controle durante Teste

Por exemplo, suponha que o período do processo de controle é igual a 2. Inicialmente, o processo repete a cada 2 unidades de tempo, o que implica na sua ativação nos instantes 2,4,6,8, etc. Porém, depois de sair da região proibida [3,5] no instante 5, o processo deve recomeçar suas repetições a partir desse instante, ou seja, nos instantes 5,7,9,11, etc. Se a renovação não for feita, o processo só seria ativado novamente no instante 6 o que causaria uma perda de controle do processo no sistema.

4.2.2.5. LEMAS E TEOREMAS

Teorema 2

O problema de decidir se é possível escalonar um conjunto de processos periódicos que usam semáforos para realizar a exclusão mútua é um problema NP_Hard.

Observação

Durante as análises e estudos de algoritmos de escalonamentos, surgiram muitos problemas importantes que foram provados como sendo NP_Complete ou NP_Hard. Portanto esse assunto merece uma descrição mais clara e objetiva para o melhor entendimento dos problemas discutidos.

Normalmente um problema é especificado com a descrição geral dos seus parâmetros e o tipo de solução satisfatória. A instância de um problema é obtida ao especificar valores particulares de todos os parâmetros do problema. Um algoritmo é dito "poder resolver um problema" se, aplicando o algoritmo em qualquer instância do problema, ele sempre produzir solução para ela.

Além disso, define-se a **função de complexidade de tempo** de um algoritmo como sendo uma função de tempo necessário, com a qual o algoritmo resolve uma instância do problema, em relação ao tamanho da instância referida. Assim, um **algoritmo de tempo polinomial** é aquele, cuja função de complexidade de tempo é uma função polinomial, e um **algoritmo de tempo exponencial** tem como função de complexidade de tempo uma função exponencial. A diferença significativa entre esses dois tipos de algoritmos é observada quando as intâncias do problema são grandes. A tabela abaixo ilustra a comparação de tempo dos algoritmos (polinomial X exponencial) em

função do tamanho da entrada do problema.

OBS : n ⟶ tamanho de entrada do problema

	n = 10	n = 20	n = 30	n = 40	n = 50	n = 60
Função Polinomial n^2	0.0001 seg.	0.0004 seg.	0.0009 seg.	0.0016 seg.	0.0025 seg.	0.0036 seg.
Função Exponencial 2^n	0.001 seg.	1.0 seg.	17.9 minutos	12.7 dias	35.7 anos	366 séculos

Sendo assim, um problema é chamado **intratável** se é tão difícil de encontrar uma solução que nenhum algoritmo de tempo polinomial consiga possivelmente resolvê-lo. Um problema de decisão é aquele que requer uma ressposta "sim" ou "não". Define-se a **classe NP de problemas** como sendo uma classe de problemas onde cada um deles pode ser resolvido pelo computador **não-determinístico** em tempo polinomial. Muitos **problemas de decisão intratável** encontrados na prática pertencem à essa classe.

Usando as definições básicas citadas acima, define-se então o problema **NP_Complete**. Segundo a teoria de Cook [COO 71], existe um problema particular na classe NP denominado **"Satisfatoriedade"** que possui a propriedade de que qualquer outro problema em NP pode ser reduzido a ele mesmo. Portanto, se o problema de "Satisfatoriedade" pode ser resolvido por um algoritmo de tempo polinomial então qualquer problema em NP também o pode. Porém, se qualquer problema em NP é intratável, o problema de "Satisfatoriedade" deve ser intratável também. Nesse caso, o problema de "Satisfatoriedade" é o problema mais difícil em NP. Assim, devem existir outros problemas em NP que possuem essa mesma propriedade de ser o mais difícil como o da "Satisfatoriedade". Desta forma, para uma grande variedade de problemas em NP foi provada a equivalência de dificuldade a esses problemas; chama-se essa classe de problemas como **NP_Complete**. Garey [GAR 79] propôs uma série de técnicas para provar a característica de NP_Complete de um problema. Posteriormente, ele garante que essas técnicas podem ser aplicadas para provar que outros problemas fora do domínio de NP também podem ser tão difíceis quanto a classe de NP_Complete.

Desse modo, qualquer problema, independentemente de pertencer à

classe NP, que pode ser transformado (usando uma função de transformação) num problema de NP_Complete terá então a mesma propriedade de "não poder ser resolvido em tempo polinomial a menos que o algoritmo de tempo polinomial seja encontrado". Definem-se esses problemas como membros da classe de **NP_Hard**.

Na verdade, a questão do problema de NP_Complete ser intratável é considerado uma das mais abertas e discutíveis na matemática contemporânea e nas ciências de computação. Apesar da disposição de muitos pesquisadores de conjeturar o problema de NP_Complete ser intratável, poucos progressos foram feitos no sentido de estabelecer uma prova ou refutação desta conjetura. De qualquer modo, mesmo sem a prova de que NP_Complete implica em intratabilidade, um problema reconhecido como NP_Complete dá indicação da necessidade de um grande esforço para resolvê-lo com o algoritmo de tempo polinomial.

Teorema 3

Se existe um escalonamento viável para um conjunto de processos onde são realizadas trocas de mensagens sincronizadamente (rendezvous) e executadas certas atividades mutuamente exclusivas, então o escalonador do monitor "kernelized" pode ser usado para encontrá-lo.

CAPÍTULO 5

CONCLUSÃO

5.1. RESULTADOS DA ANÁLISE

5.2. COMPARAÇÃO ENTRE "EARLIEST DEADLINE" E "TAXA MONOTÔNICA"

5.3. DESEMPENHO DAS FERRAMENTAS

5.4. SUGESTÕES

5.5. CONSIDERAÇÕES FINAIS

5. CONCLUSÃO

A idéia inicial deste trabalho foi a de realizar análises e testes com os algoritmos que resolvem problemas de escalonamento em ambientes tais como mostrados no diagrama da figura 2.1. O objetivo foi alcançado no decorrer dos estudos que tiveram como motivação investigar problemas adicionais dos próprios algoritmos e conceber uma série de propostas para a implementação de um algoritmo num núcleo convencional.

5.1. RESULTADOS DA ANÁLISE

Para o escalonador estático "Taxa Monotônica", a escalonabilidade de um conjunto de processos periódicos e independentes pode ser pressuposta através de uma condição suficiente mas não necessária. Já para um conjunto de processos periódicos e dependentes, pode ocorrer o vencimento do prazo de alguns processos se a inversão de prioridade não for prevenida com os devidos cuidados. Assim, o protocolo de herança de prioridade foi usado para resolver esse problema; porém, o mesmo implica efeitos colaterais tais como "deadlock" e bloqueio múltiplo. O aperfeiçoamento deste levou ao protocolo de prioridade topo que, por sua vez, possibilitou a formulação de várias condições suficientes que permitem testar a viabilidade do escalonador para um conjunto de processos periódicos e dependentes. Tanto o protocolo de herança de prioridade, quanto o protocolo de prioridade topo, funcionam através de testes adicionais durante a entrada e a saída das regiões críticas, que geralmente são incorporados nos comandos WAIT e SIGNAL do núcleo. Contudo, para tornar esses protocolos funcionalmente práticos, surgiu a idéia de criar uma "casca" sobre o núcleo de modo que possam ser aplicados a nível do usuário em núcleos convencionais.

Num ambiente onde o escalonador estático, além de cuidar das execuções dos processos periódicos, também deve atender os processos aperiódicos que chegam ao sistema em instantes aleatórios, a necessidade de um servidor eficiente para auxiliar o escalonador é essencial para manter o bom desempenho do mesmo e os servidores Background, Polling e DS foram utilizados para este fim. A análise foi feita em função de comparações entre eles e discussões sobre as vantagens e desvantagens de cada um. A princípio, um servidor de processos aperiódicos pode ser considerado como um processo normal, cedendo seu tempo de execução para os aperiódicos através de algumas adaptações. Apesar dos tempos adicionais (overhead), que são inevitáveis, as propostas para a implementação de servidores oferecem uma ótima

alternativa, no caso em que as restrições temporais dos processos periódicos devem ser cumpridas sem atrasar muito o tempo de resposta dos processos aperiódicos.

No caso do escalonador dinâmico "Earliest Deadline", um conjunto de processos periódicos e independentes pode ser verificado antes da execução, quanto à sua escalonabilidade, através de uma condição suficiente e necessária. Logo, a análise direciona aos processos dependentes os quais compartilham área comum ou realizam trocas de mensagens pelo mecanismo rendezvous.O uso das primitivas de rendezvous gera uma relação de precedência entre os processos comunicantes, o que pode provocar a perda dos prazos dos processos. Assim, realizou-se a análise da técnica de "Prazos Revisados" que providencia um escalonamento off-line que examina e calcula os prazos dos blocos das instâncias ocorridas num determinado intervalo. Uma vez que este intervalo é adotado como sendo o mínimo múltiplo comum dos períodos dos processos, a técnica se torna menos exaustiva se os períodos dos processos não forem primos entre si. Todavia, a técnica "Prazos Revisados", proposta por Mok [MOK 83], mostrou suas ineficiências quando é aplicada em vários conjuntos de processos, sendo assim, aprofundamos as análises no sentido de encontrar solução para cobrir suas falhas. Logo, uma verificação final, proposta pela autora, é usada para completar a técnica referida, garantindo assim a previsibilidade de escalonamento de um conjunto de processos comunicantes (com rendezvous).

Quando os processos usam regiões comuns, uma maneira de protegê-las é fixar uma unidade básica de execução do núcleo, chamada quantum, com o tamanho da região compartilhada. Assim, os processos são divididos e executados em cada quantum sem interrupção. Tendo em vista esta estrutura, a técnica "Região Proibida" foi empregada para garantir o não vencimento dos prazos e sua funcionalidade torna a ter sentido, quando as regiões críticas são pequenas.

Ambas as técnicas são aplicadas aos processos, antes destes serem executados, gerando então uma base de informações a qual é consultada durante o sistema pelo escalonador.

5.2. Comparação entre "Earlieste Deadline" e "Taxa Monotônica"

No escalonamento de conjuntos de processos periódicos e independentes, o "Earliest deadline" apresenta seu menor limite superior U_{MS} igual a 1 que é maior ou igual ao do "Taxa monotônica", $U_{MS} = m * (2^{1/m}-1)$. Lembrando o conceito citado por Liu [LIU 73] que, o fator de utilização de CPU, U, de um conjunto de processos deve ser menor ou igual a seu menor limite superior de utilização de

CPU, U_{MS}, para que este possa ser escalonável. Então, suponhamos U_x, o fator de utilização de CPU de um conjunto de x processos. Se $U_x \leq x * (2^{1/x}-1)$ então ele é escalonável pelo "Taxa Monotônica" e consequentemente pelo "Earliest Deadline", pois $U_x \leq x * (2^{1/x}-1) \leq 1$. Mas, se $U_x > x * (2^{1/x}-1)$ então ele não é escalonável pelo "Taxa Monotônica" e pode ser escalonável pelo "Earliest Deadline" se $U_x \leq 1$. Concluímos então que "Earliest Deadline" consegue escalonar mais conjunto de processos independentes e periódicos que o "Taxa Monotônica".

Porém, para aqueles processos que são dependentes a situação se modifica. Pela própria característica do escalonador estático, a atribuição fixa de prioridades permite várias análises temporais dos processos; por exemplo, a análise do tempo de bloqueio que é um fator importante durante o cálculo da condição que visa determinar, antes do funcionamento do sistema, a escalonabilidade de um conjunto de processos que compartilham a região comum. É possível ainda efetuar a mesma verificação para um conjunto de processos que possuem seu prazo menor que o período, desde que a diferença entre o prazo e o período esteja incluída no tempo de bloqueio do processo. Em contraste ao escalonador estático, a atribuição de valores inconstantes de prioridades aos processos do "Earliest Deadline" dificulta essa exploração no sentido de presumir a viabilidade de escalonamento de um conjunto de processos dependentes. Entretanto, foi adotado um esquema de escalonamento off-line para providenciar algumas verificações e adaptações; os dados produzidos são fornecidos ao núcleo do sistema para serem usados posteriormente pelo escalonador on-line. No caso, o tempo de bloqueio de cada processo é imprevisível.

5.3. Desempenho das Ferramentas Usadas

Em relação às ferramentas usadas, o núcleo de tempo real DEADMOSI teve uma contribuição importante na visualização e vivência realística e de processos concorrentes num HRTS; também foi responsável pela grande maioria dos testes dos algoritmos analisados. Apesar de seu uso tornar as análises possíveis, o longo tempo que decorre durante os testes dos algoritmos afeta o sentido ideal da simulação, principalmente quando um conjunto grande de processos é considerado no sistema. Além do mais, o tempo real da execução no corpo do processo não é encontrado com a precisão desejada em relação ao valor definido do parâmetro c_i. Apesar disso, sua grande utilidade no trabalho foi algo gratificante.

Já o SSE, apesar do fato de ter sido concluído apenas na fase final deste trabalho, pela sua característica de simulador, mostrou grande eficiência

nos testes. No futuro, ele deve desempenhar um grande auxílio na exploração dos estudos e análises análogos à deste trabalho.

5.4. SUGESTÕES

Uma série de questões envolvidas nos algoritmos de escalonadores em diferentes ambientes foram levantadas e discutidas durante o trabalho; contudo, existem ainda alguns pontos abertos que podem conduzir futuramente a análises interessantes.

▸ SUG 1 : No escalonador dinâmico, um conjunto de processos periódicos independentes pode ter prevista sua escalonabilidade através da condição suficiente e necessária

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

com a condição de que o prazo de cada processo coincida com o tamanho do seu período. Caso o processo possua prazo menor que o do seu respectivo período, pode então haver uma condição semelhante para o mesmo fim.

▸ SUG 2 : O problema da região crítica foi contornado, no escalonador "Earliest Deadline", usando a idéia de "Monitor Kernelized" e a técnica "Região Proibida"; porém, estes são restritos para um conjunto de processos que possua região comum pequena. O protocolo de prioridade topo aplicado no escalonador estático é mais flexível em relação ao tamanho da região crítica e ainda possibilita uma condição para pré-validar um conjunto de processos. Assim, é interessante adaptar o protocolo ao escalonador dinâmico.

▸ SUG 3 : Como foi destacado no capítulo 4, as localizações dos pontos de comunicação rendezvous influenciam na escalonabilidade de um conjunto de processos dependentes. Uma análise mais detalhada sobre isso pode levar a conclusões interessantes.

5.5. Considerações Finais

O HRTS está se tornando uma área aberta de pesquisa que por sua vez apresenta desafios computacionais tais como os tratamentos de restrição temporal. O presente trabalho percorreu os primeiros passos nessa direção, visando estudos em dois principais escalonadores num ambiente monoprocessado, e pode ser visto como um incentivo para a pesquisa na área. Visto que o escalonamento de processos dependentes em HRTS não é um problema trivial, optou-se então por um ambiente de um único processador para uma investigação inicial. Uma vez dominado esse problema, o próximo passo é a expansão da análise aos sistemas multiprocessadores e sistemas distribuídos.

[RAJ 88] Rajkumar, R., Sha, L. and Lehoczky, J. P.; "Real-Time Synchronization Protocol for Multiprocessors". *Proc. IEEE Real Time Systems Symp., Los Angeles, California, pp.259-269,* 1988

[SHA 87] Sha, Lui, Rajkumar, R. and Lehoczky, J.P.; "Priority Inheritance Protocols : An Approach to Real Time Synchronization". *Technical Report, Department of Computer Science, CMU,* 1987.

[SHA 89] Sha, Lui et al; "Mode Change Protocols for Priority-Driven Preemptive Scheduling". *J. Real-time Systems, vol 1, pp.243-264,* 1989

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

[SPA 91] Spanó, Rodrigo M.; "Sistema de Simulação para Escalonadores de Tempo Real Crítico"". *Dissertação de Mestrado, FEE - Unicamp,* Abril 1991

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

[STA 88] Stankovic, John A.; "Misconceptions About Real-Time Computing". *Computer, pp.10-19,* October 1988.

[ZHA 87] Zhao, Wei, Ramamritham, Krithivasan and Stankovic, J. A.; "Scheduling Tasks with Resource Requirements in Hard Real Time System". *IEEE Trans. Software Eng., vol se 13, No 5,* May 1987.

[YOU 82] Young, Stephen J. ; "Real Time Languages : Design and Development". *Ellis Horwood Ltd. ,* 1982.