EXCLUSÃO MÚTUA NA AUSÊNCIA DE REGISTRADORES ATÔMICOS

Luiz Claudio Rosa e Silva Maia

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovado por:

Prof. Valmir Carneiro Barbosa, Ph. D.
(presidente)

Prof. Edil Severiano T. Fernandes, Ph. D.

Prof. Adriano Joaquim de O. Cruz, Ph. D.

RIO DE JANEIRO, RJ - BRASIL

FEVEREIRO DE 1991

MAIA, LUIZ CLAUDIO ROSA E SILVA

Exclusão Mútua na Ausência de Registradores Atômicos [Rio de Janeiro] 1991

VI, 102, 29.7 cm, (COPPE/UFRJ, M. Sc., ENGENHARIA DE SISTEMAS E COMPUTAÇÃO, 1991)

TESE - Universidade Federal do Rio de Janeiro, COPPE

1- Controle de concorrência 2- Primitivas não-atômicas

3- Exclusão-ℓ

I. COPPE/UFRJ II. Título (Série).

Resumo da Tese apresentada à COPPE como parte dos requisitos necessários para obtenção do grau de Mestre de Ciências (M. Sc.).

Exclusão Mútua na Ausência de Registradores Atômicos

Luiz Claudio Rosa e Silva Maia

Fevereiro de 1991

Orientador: Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Neste trabalho estudamos o comportamento de um sistema concorrente baseado em primitivas de sincronização não-atômicas. Tais primitivas são constituídas de registradores mais simples que os atômicos e podem ser acessadas simultaneamente. O modelo experimental escolhido para esta análise foi o problema clássico de Exclusão-ℓ. Este problema foi solucionado utilizando primitivas baseadas em três tipos de registradores diferentes, de modo a permitir uma análise comparativa do desempenho do sistema em diversas situações práticas. Finalmente, traçamos conclusões sobre o desempenho das soluções e identificamos as condições em que a utilização de cada registrador é mais adequada.

Abstract of the Thesis presented to COPPE as partial fulfillment of the requirements for the degree of Master of Science (M. Sc.).

Mutual Exclusion in the Absence of Atomic Registers

Luiz Claudio Rosa e Silva Maia

February, 1991

Thesis Supervisor: Valmir Carneiro Barbosa

Department: Programa de Engenharia de Sistemas e Computação

In this work, we study the behavior of a concurrent system based in non-atomic primitives. Such primitives are based in registers weaker than atomic and allow simultaneous access. The model chosen for this analysis was the classic ℓ-Exclusion Problem. This problem was solved using primitives based on three different types of registers, in order to allow a comparative analysis of system's performance in several practical situations. Finally, we draw conclusions about the performance of the solutions and identify the conditions on which the use of each register is more adequate.

ÍNDICE

CAPÍTULO I

INTRODUÇÃO

I.1 O PROBLEMA

Até o presente momento, a grande maioria das pesquisas feitas em controle de concorrência de processos foi baseada em algum tipo de primitiva de sincronização atômica.

Entretanto, estas operações atômicas são excessivamente poderosas e ao serem utilizadas para solucionar questões de concorrência, mascaram a real dificuldade do problema. Como conseqüência, estas primitivas obrigam os processos mais rápidos a esperarem que os mais lentos concluam suas operações, levando assim a um comportamento ineficiente.

O *test-and-set* é um exemplo deste tipo de primitiva de sincronização poderosa. Esta primitiva impõe exclusão mútua aos processos que a operam concorrentemente e por esta razão, pode ser usada para serializar operações concorrentes, tornando-as atômicas. A maioria dos computadores atuais oferece estas primitivas de sincronização sofisticadas diretamente implementadas por *hardware*, o que facilita a difusão do seu uso em aplicações concorrentes.

Em 1986, LAMPORT (em [2]) desenvolveu um novo formalismo para o tratamento de sistemas concorrentes e introduziu uma nova solução para o problema clássico de exclusão mútua baseado nestes novos conceitos. Até aquela data, todos os modelos formais de concorrência de processos eram baseados no conceito de operações atômicas indivisíveis. Por esta razão, a solução apresentada tem uma grande importância teórica, pois resolve o problema diretamente através do uso de registradores acessados simultaneamente, dispensando a etapa intermediária de

obtenção da atomicidade.

LAMPORT deu um importante passo ao evitar o uso de operações de sincronização atômicas ao introduzir um mecanismo muito fraco de comunicação em memória compartilhada, conhecido como *safe bit* **SWMR** ("Single Writer Multiple Readers"). Uma operação de leitura neste *bit* retorna o valor correto (o último a ser escrito) apenas se a operação não for concorrente com nenhuma operação de escrita, caso contrário retorna aleatoriamente 0 ou 1.

Como este formalismo é muito recente, não existem aplicações práticas que nos permitam avaliar o comportamento de sistemas de processos paralelos baseados nesta nova teoria. Intuitivamente podemos esperar que os sistemas concorrentes baseados em primitivas não-atômicas sejam mais eficientes, uma vez que é eliminada a espera em caso de operações simultâneas. Entretanto, os algoritmos que implementam este novo método de tratamento de concorrência são mais complexos e exigem um tempo de processamento maior para serem executados.

I.2 ANÁLISE EXPERIMENTAL

O objetivo desta tese é justamente analisar o desempenho de sistemas concorrentes que, baseados nesta nova teoria, dispensam o uso de primitivas atômicas, e compará-los aos resultados obtidos através dos métodos convencionais que necessitam da atomicidade.

Para esta análise comparativa foi escolhido um exemplo clássico de controle de concorrência, conhecido como o problema de Exclusão-ℓ, que foi introduzido e resolvido por FISCHER, LYNCH, BURNS, e BORODIN em [6]. Este problema pode ser considerado como um problema de alocação de recursos em que existem ℓ instâncias idênticas de um recurso não-compartilhável e cada processo pode requisitar no máximo uma

instância deste recurso. É uma generalização do problema de exclusão mútua descrito e resolvido por DIJKSTRA em [8], consistindo de um conjunto de processos assíncronos que executam alternadamente uma seção crítica e uma não-crítica, e que devem ser sincronizados de modo que no máximo ℓ processos executem suas seções críticas simultaneamente.

Em 1988, DOLEV, GAFNI e SHAVIT (em [1]) publicaram uma solução para o problema de Exclusão-ℓ baseada no formalismo desenvolvido por LAMPORT, resolvendo o controle de concorrência diretamente através de *safe bits*, sem a necessidade de primitivas de sincronização atômicas. Foram criadas novas estruturas simples de sincronização utilizando registradores *safe*, capazes de substituir a atomicidade em aplicações concorrentes e de resolver vários aspectos de problemas de sincronização. A solução obtida com estas estruturas abstratas é determinística e satisfaz as propriedades de Exclusão-ℓ, ausência de *Deadlock* e ausência de *Lockout*.

Para fins de comparação de comportamento, o algoritmo proposto por DOLEV, GAFNI e SHAVIT, inicialmente desenvolvido para utilizar registradores *safe*, foi simplificado de modo a utilizar outros tipos de registradores mais fortes. Desta forma, foram desenvolvidos três programas, sendo que as implementações mais complexas correspondem aos registradores mais fracos.

Os algoritmos foram implementados e executados em um microcomputador com arquitetura compatível com a do IBM PC AT, possuindo *chips* de memória atômica com células de 8 *bits*.

O registradores **SWMR** foram simulados por rotinas que implementam suas operações de leitura e escrita utilizando células de memória de 8 *bits* acessadas por todos os processos. Os tempos de acesso a estes registradores simulados foram projetados para serem iguais no caso de

operações não-concorrentes, de modo a simplificar a análise comparativa dos resultados. No caso de operações concorrentes, o tempo de acesso dos registradores atômicos simulados é maior, pois as operações devem ser serializadas, ou seja, executadas uma de cada vez.

A simulação da execução do sistema por vários processadores separados, foi obtida através do desenvolvimento de um núcleo para o sistema (*kernel*), com a função de repartir o tempo de processamento entre os vários processos.

Para analisar o desempenho da solução, foram feitas medidas variando-se alguns parâmetros do problema, como o número de processos no sistema, o número de recursos e os tempos de execução gastos pelo processamento das seções críticas e não-críticas dos programas dos processos. Como resultado da execução do sistema, obtivemos um índice de sua eficiência, dado pela razão entre o número médio de vezes que cada processo executa sua seção crítica e seu tempo de processamento. Estas medidas experimentais foram repetidas para os três tipos de registradores e os resultados posteriormente apresentados na forma de gráficos para facilitar sua visualização e análise comparativa.

I.3 CONCLUSÕES

Os resultados obtidos experimentalmente mostram que o algoritmo para o problema de Exclusão-ℓ, baseado em registradores atômicos, apresenta um desempenho melhor na maioria dos casos analisados.

Apesar dos registradores *safe* e regular não imporem nenhum tipo de espera entre os processos em caso de operações simultâneas, seus algoritmos são mais complexos, requerendo um número maior de *bits* de comunicação e conseqüentemente um maior número de operações sobre estes *bits*. Os algoritmos baseados nestes tipos de registradores são mais eficientes nos casos em que o grau de concorrência nos registradores de

comunicação é alto, ou seja, quando o número de operações de leitura e escrita simultâneas é grande comparado com o número total destas operações. O fato da análise comparativa favorecer a solução do problema de Exclusão-ℓ com registradores atômicos pode ser justificado pela baixa percentagem de operações concorrentes observada na maioria das situações testadas (2 a 6%).

Através da comparação dos resultados de cada um dos algoritmos, podemos concluir que os algoritmos baseados em registradores *safe* e regulares, são mais adequados nas situações em que o número de operações concorrentes de leitura e escrita nos registradores compartilhados é alto em relação ao total destas operações. Nos casos em que este número é baixo, a solução baseada em registradores atômicos é mais adequada.

I.4 DESCRIÇÃO DOS CAPÍTULOS

O capítulo II descreve os tipos de registradores utilizados no mecanismo de comunicação entre processos e seus comportamentos em caso de concorrência. É também descrita a arquitetura e analisado o funcionamento de memórias atômicas do tipo *multi-porta*

O capítulo III introduz o problema de Exclusão-ℓ e apresenta a solução proposta por DOLEV, GAFNI e SHAVIT. São apresentados algoritmos que satisfazem as propriedades de ausência de *deadlock* e de *lockout*, assim como as novas estruturas de dados que eliminam a necessidade de primitivas de sincronização atômicas.

O capítulo IV descreve o método utilizado na implementação prática da solução do problema de Exclusão-ℓ, assim como a adaptação do algoritmo, de forma a utilizar os outros tipos de registradores. É descrito também o método de simulação destes registradores a partir de células de memória atômicas. Finalmente, é feita uma análise das

variáveis do problema e apresentado o conjunto de condições ao qual os algoritmos foram submetidos para a obtenção dos índices de desempenho.

O capítulo V apresenta a análise dos resultados experimentais juntamente com algumas justificativas para o comportamento dos algoritmos nas várias condições a que o sistema foi submetido. Por último, são comparados os resultados de desempenho obtidos pelos três algoritmos implementados e apresentadas as conclusões sobre comportamento das soluções que não necessitam de primitivas atômicas.

O formalismo desenvolvido por LAMPORT para tratamento de sistemas concorrentes, no qual se baseia esta tese, pode ser encontrado em [2] e [4].

CAPÍTULO II

TIPOS DE REGISTRADORES

II.1 CLASSIFICAÇÃO

Para efetuar o controle de concorrência entre processos assíncronos, é necessário algum tipo de comunicação entre os processos. Todo tipo de comunicação envolve uma via cujo estado é alterado pelo transmissor e observado pelo receptor. Este estado pode ser, por exemplo, a voltagem em um fio ou então uma mensagem.

O método de comunicação adotado nesta tese é conhecido como registrador compartilhado. Neste caso, os processos transmissor e receptor são chamados de escritor e leitor, respectivamente, e o estado do meio de comunicação é conhecido como o valor do registrador.

Nesta tese nos restringimos à comunicação unidirecional, em que um único processo pode modificar o valor do registrador. Para permitir a comunicação bidirecional são necessários dois *bits* de comunicação, cada um modificado por um dos processos. Não há restrições ao número de processos que pode ler o registrador.

Os registradores que satisfazem estas características são denominados **SWMR** ("Single Writer Multiple Readers") e possuem um único escritor e vários leitores.

Como o escritor é único, não existe a possibilidade de ocorrerem operações simultâneas de escrita em um mesmo registrador. É suposto que o resultado das leituras não é afetado por outras leituras concorrentes. Como o leitor apenas "sente" o valor armazenado no registrador, não há razão para que a leitura interfira com outra operação de leitura ou de escrita.

Para classificar os registradores é preciso analisar o

comportamento dos mesmos no caso de operações de leitura concorrentes com uma ou mais operações de escrita. Nesta condição existem três possibilidades para o comportamento dos registradores, o que nos permite dividí-los em três tipos, de acordo com os resultados retornados pelas leituras concorrentes. São eles:

1) Registrador **SAFE** - É o caso mais fraco. Este tipo de registrador retorna o valor correto, ou seja, o último valor escrito, para toda operação de leitura que não for concorrente com nenhuma operação de escrita. Nada podemos afirmar sobre o valor obtido por uma operação de leitura concorrente com uma operação de escrita. Assim, se um registrador *safe* pode armazenar os valores 1, 2 e 3, por exemplo, então qualquer operação de leitura irá obter um destes três valores. Uma operação de leitura concorrente com uma de escrita que altera o valor armazenado de 1 para 2, poderá retornar qualquer um destes valores, inclusive 3.

2) Registrador **REGULAR** - É um registrador *safe* em que uma operação de leitura concorrente com uma de escrita retorna o valor antigo ou o novo. Desta forma é retornado o valor correto para operações de leitura que não sejam concorrentes com nenhuma operação de escrita. Por exemplo, uma operação de leitura concorrente com uma de escrita que altera o valor armazenado de 1 para 3, pode retornar 1 ou 3, mas não 2. Podemos dizer então, que uma leitura concorrente com uma série de operações de escrita obtém o valor inicial (anterior à primeira escrita) ou um dos valores escritos.

3) Registrador **ATÔMICO** - É o caso mais forte. É um registrador *safe* em que as operações de leitura e escrita se comportam como se

fossem executadas em uma determinada ordem. Podemos dizer que existe maneira de ordenar totalmente as operações de leitura e escrita de modo que os resultados retornados sejam os mesmos que os retornados caso as operações tivessem sido executadas nesta ordem, mas sem sobreposição (concorrência).

As diferenças entre os registradores podem ser melhor compreendidas através do exemplo dado pela figura II.1. Nesta figura são mostradas algumas operações de leitura e escrita em registradores que podem armazenar os valores 5, 6 e 27. A duração de cada operação é indicada pelo comprimento dos segmentos de reta, sendo que o tempo corre da esquerda para a direita. Uma operação de escrita do valor 5 precede todas as outras operações. Em seguida são feitas três operações de leitura denominadas $leitura_1$, $leitura_2$, e $leitura_3$.

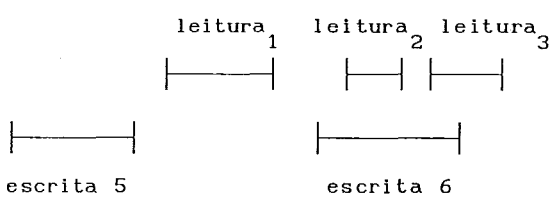


Figura II.1

Executando as operações acima em registradores *safe* obteremos o seguinte resultado:

$Leitura_1$= 5 , pois não é concorrente com nenhuma operação de escrita.

$Leitura_2$ e $Leitura_3$ podem retornar qualquer um dos valores possíveis, ou seja, 5, 6 ou 27, pois são concorrentes com a segunda operação de escrita.

Em registradores regulares temos:

$Leitura_1$= 5 , pois não é concorrente com nenhuma operação de escrita. $Leitura_2$ e $Leitura_3$ podem retornar os valores 5 (antigo) ou 6 (novo), mas não 27. Em particular pode ocorrer o caso em que a $leitura_2$ retorne 6 e a $leitura_3$ retorne 5.

As mesmas operações executadas em registradores atômicos obteem os seguintes resultados:

$Leitura_1$= 5 , pois não é concorrente com nenhuma operação de escrita.

As outras duas operações de leitura podem obter uma das combinações abaixo:

$Leitura_2$	$Leitura_3$
5	5
5	6
6	6

O segundo par de valores desta tabela (5,6) representa a situação em que as operações se comportam como se a primeira leitura ($leitura_2$) precedesse a escrita e a segunda ($leitura_3$) ocorresse logo após a mesma escrita. Podemos observar também, que ao contrário dos registradores regulares, os registradores atômicos não permitem a possibilidade da $leitura_2$ obter o valor 6 (novo) e a $leitura_3$ o valor 5 (antigo).

Em geral, se duas leituras sucessivas se sobrepõem à mesma escrita, o registrador regular permite que a primeira leitura obtenha o valor novo enquanto que a segunda retorna o valor antigo. Este comportamento é proibido em registradores atômicos.

Assim, podemos concluir que um registrador regular é também

atômico se duas operações de leitura sucessivas nunca forem concorrentes com uma mesma operação de escrita.

II.2 REGISTRADORES COM UM ÚNICO ESCRITOR

Os registradores de um único escritor são classificados de acordo com as seguintes características:

1) Quanto aos requerimentos satisfeitos pelos registradores
   - SAFE, REGULAR ou ATÔMICO
2) Quanto ao valor que pode ser armazenado no registrador
   - BOLEANO ou MULTI-VALOR
3) Quanto ao número de leitores
   - ÚNICO-LEITOR ou MULTI-LEITOR(**SWMR**)

Estas opções geram 12 classes diferentes de registradores, sendo que o mais simples e de mais fácil implementação é o *safe*, booleano com um único leitor. Este é o mais adequado para ser implementado com a tecnologia atual, e requer apenas que o escritor estabeleça um nível de tensão (alto ou baixo) e que o leitor possa testá-lo sem alterá-lo.

Todas as outras classes de registradores podem ser implementadas a partir de registradores de classes mais fracas, com exceção do registrador atômico multi-valor, que continua como um problema aberto, como é demonstrado em [3].

II.3 - IMPLEMENTAÇÃO DE MEMÓRIAS ATÔMICAS

No estágio atual da tecnologia, a implementação mais comum dos registradores atômicos é representada pelos *chips* de memória multi-porta. Este tipo de memória possui vários barramentos independentes de dados e endereço. Apesar disto, as operações de

leitura e escrita nesta memória não são concorrentes, mas sincronizadas internamente de modo a haver uma serialização das operações. Assim, se dois processos tentam acessar simultaneamente uma mesma célula de memória, a lógica de controle interna do *chip* faz com que um dos processos espere até que o outro conclua a sua operação.

Um *chip* de memória *dual-port* possui dois barramentos separados de endereço e dois de dados. Não existem restrições para as operações de leitura e escrita executadas em posições (células) diferentes da memória.

As operações concorrentes em uma mesma célula são serializadas, sendo que uma delas é atrasada até a conclusão da outra. Esta serialização é imposta pela lógica de arbitragem interna do *chip* e necesssita de um tempo para ser executada. Em memórias *dual-port* comerciais, este tempo de chaveamento é aproximadamente igual ao tempo de acesso da memória. A Figura II.2 abaixo apresenta um diagrama simplificado de uma memória *dual-port*.

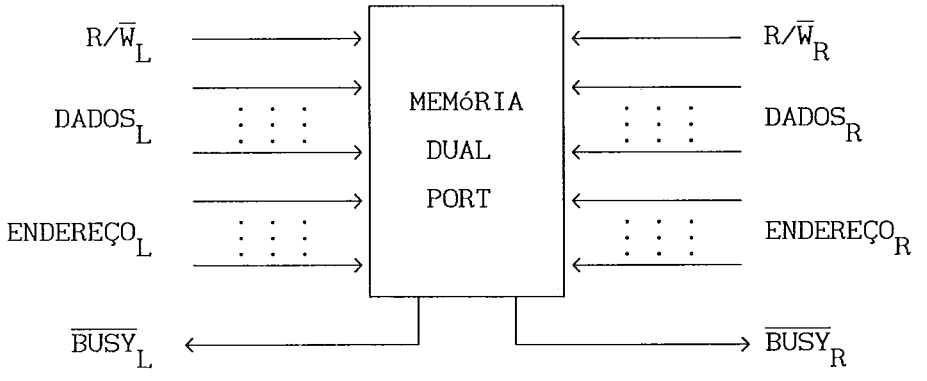
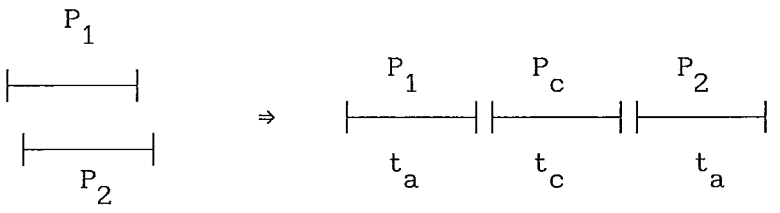


Figura II.2 - Esquema de um chip de memória dual-port

Desta forma, as operações concorrentes P_1 e P_2 executadas em uma mesma posição de memória atômica se comportam como mostrado abaixo:



onde t_a = tempo de acesso da memória

t_c = tempo de chaveamento da lógica de arbitragem

P_1 e P_2 = operações de leitura ou escrita na mesma célula

P_c = operação de chaveamento interna do *chip*

Figura II.3

Podemos observar no exemplo da figura II.3, que o tempo gasto para execução concorrente da operação P_2 em memórias atômicas é $t_{at} = 2t_a + t_c \approx 3t_a$. Esta mesma operação realizada em memórias do tipo *safe* e regular seria concluída em um tempo t_a, pois as operações podem ser realizadas simultaneamente.

CAPÍTULO III

O PROBLEMA DE EXCLUSÃO-L

III.1 INTRODUÇÃO

O problema de exclusão mútua foi originalmente descrito e resolvido por DIJKSTRA em [8]. Este problema consiste de um conjunto de processos assíncronos, cada qual executando alternadamente uma seção crítica e uma não-crítica, que devem ser sincronizados de modo que dois processos nunca executem suas seções críticas concorrentemente. Várias outras soluções se sucederam à solução de DIJKSTRA, sendo a maioria baseada em primitivas de sincronização fornecidas pelo *hardware* e visando a sua aplicação prática. Em [4], LAMPORT apresenta uma solução de grande importância teórica para o problema dispensando tais recursos de *hardware* e resolvendo-o diretamente através de registradores *safe* acessados simultaneamente.

De forma geral, a solução para o problema de exclusão mútua consiste na especificação de procedimentos para a sincronização dos processos de modo a controlar o acesso à sua seção crítica. Estes procedimentos de sincronização também podem ser vistos como a manipulação de um recurso comum, sendo que a alocação deste recurso corresponde à obtenção do acesso à seção crítica do processo, e a liberação corresponde à saída do processo da sua seção crítica.

Para implementar a exclusão mútua, algumas operações de sincronização devem ser incluídas em cada um dos processos, como mostrado no algoritmo a seguir:

```
declaração inicial;
repetir para sempre
  Restante;
```

entrada;
Seção crítica;
saída;
fim da repetição;

A declaração inicial corresponde ao procedimento de inicialização das variáveis de comunicação dos processos.

A declaração de entrada é a seção do código executada pelo processo antes de entrar na sua seção crítica. Tem como função sincronizar o acesso dos processos a esta seção. Podemos dizer que esta parte do código aloca o recurso correspondente à seção crítica.

A declaração de saída é a seção do código que é executada pelo processo ao deixar a sua seção crítica e retornar para a Restante, o que equivale à liberação do recurso correspondente à sua seção crítica.

A solução do problema consiste na especificação das três declarações acima, de forma a satisfazer a propriedade de exclusão mútua.

É desejável que as propriedades de ausência de *deadlock* e de ausência de *lockout* também sejam satisfeitas pela solução. *Deadlock* é o estado em que todos os processos em um sistema estão esperando por um evento que só pode ser causado por outro processo no sistema. No caso particular do problema de exclusão mútua, este evento pode ser visto como a liberação do recurso correspondente à seção crítica. Assim, a ausência de *deadlock* implica que o sistema de processos como um todo pode continuar para sempre fazendo progressos. Entretanto, não elimina a possibilidade de um processo individualmente esperar para sempre na sua seção de entrada.

A propriedade de ausência de *lockout* visa a garantir que qualquer processo que tentar entrar na sua seção crítica irá eventualmente conseguir, embora não se possa precisar quando isto

ocorrerá. Em particular, é permitido que outros processos executem suas seções críticas um número arbitrário de vezes antes que o processo i execute a sua seção crítica.

III.2 DESCRIÇÃO DO PROBLEMA

O problema de Exclusão-ℓ é uma generalização do problema de exclusão mútua. Permite que um número de processos ℓ ($\geq$ 1), mas não mais, acessem simultaneamente suas seções críticas. É considerado um problema de alocação de recursos em que existem ℓ instâncias idênticas de um recurso não-compartilhável e cada processo pode requisitar no máximo uma instância deste recurso.

A solução para este problema não deve impor restrições ao comportamento dos processos dentro das suas seções crítica e Restante, exceto que não haja comunicação com os outros processos que estão sendo executados concorrentemente.

III.3 A SOLUÇÃO

A seguir é apresentada a solução proposta DOLEV, GAFNI e SHAVIT em [1] para o problema de Exclusão-ℓ. Esta solução é determinística, justa e resolve o problema diretamente através do uso de registradores *safe*, evitando a etapa intermediária de obtenção da atomicidade.

As soluções anteriores ([6] e [7]) necessitavam da existência de uma operação sobre memórias atômicas muito mais poderosa que o *test-and-set* e tinham o objetivo de satisfazer a condições de justiça bastante rigorosas. A condição de justiça adotada em [1] visa garantir que qualquer processo que deseje utilizar um recurso eventualmente venha a conseguí-lo.

O modelo de falhas adotado, permite que os processos possam parar de forma indetectável durante a execução de qualquer operação dos

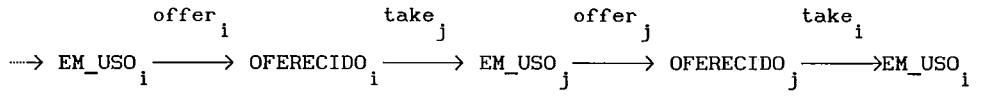
seus protocolos. Mesmo sob este tipo de comportamento, a solução deve satisfazer às exigências de Exclusão-ℓ, ausência de *Lockout* e de *Deadlock*. Entretanto, a solução não deve forçar os processos mais rápidos a esperarem que os processos lentos ou falhos terminem de executar as suas operações concorrentes.

Para se obter a Exclusão-ℓ satisfazendo às exigências descritas acima, foi necessária a criação de uma nova estrutura de dados de sincronização a partir de registradores *safe*, com a função principal de impor relações de precedência entre pares de processos. A cada par de processos está associada uma estrutura de dados composta de duas sub-estruturas simétricas (uma para cada processo no par), sendo cada uma delas composta de três *safe bits*. Estas estruturas são manipuladas pelo par de processos ao qual pertencem e lidas por todos os outros processos no sistema.

O fundamental para a sincronização é que, se um dos processos (em um par) falha, o outro ainda é capaz de manipular a estrutura de dados.

III.4 NOVAS ESTRUTURAS DE DADOS

A primeira estrutura de dados definida é o FORK. Esta estrutura é semelhante ao *fork* introduzido por CHANDY e MISRA na solução do problema *Dining Philosophers* em [9]. Assim como em [9], é utilizado para estabelecer relações de precedência entre os processos que o operam. A cada par de processos i e j está associada uma instância do FORK denominada $FORK_{ij}$. O FORK transiciona entre quatro estados: EM_USO_i, $OFERECIDO_i$, EM_USO_j e $OFERECIDO_j$, nesta ordem. A transição entre estes estados é feita através das primitivas $take_i$, $take_j$, $offer_i$ e $offer_j$ como mostrado abaixo:



O $FORK_{ij}$ pode ser observado por todos os processos através da operação $read_k$. Esta operação retorna um dos quatros estados possíveis do FORK.

III.5 IMPLEMENTAÇÃO DO FORK

O $FORK_{ij}$ é composto de um *bit* de memória compartilhada, F, e de dois *safe bits* **SWMR** (Single Writer Multiple Reader) w_i e w_j escritos por i e j respectivamente. A operação $read_k(FORK_{ij})$ é definida da seguinte forma:

```
Se  F = 1  então {
                Se w_i então retorna EM_USO_i
                   senão retorna OFERECIDO_i
}
senão {
                Se w_j então retorna EM_USO_j
                   senão retorna OFERECIDO_j
}
```

A ordem de leitura dos *bits* F, w_i e w_j é fundamental para o funcionamento da estrutura, como será demonstrado no final deste capítulo.

As operações $take_i(FORK_{ij})$ e $offer_i(FORK_{ij})$ por um alterador (processo que pode alterar o estado do FORK) i (do mesmo modo para um alterador j) são definidas como:

Take$_i$:

Se read$_k$(FORK$_{ij}$) = OFERECIDO$_j$ então

{

w_i := verdadeiro;

F := 1;

}

Offer$_i$:

Se read$_k$(FORK$_{ij}$) = EM_USO$_i$ então

w_i := falso;

Como o *bit* F é escrito por ambos os processos no par, é necessário utilizar um artifício para implementá-lo a partir de *safe bits* do tipo **SWMR**. Para isto F é definido como sendo formado por dois *safe bits*, f_i e f_j, escritos por i e j respectivamente. As operações de leitura e escrita no *bit* F são definidas da seguinte forma:

Leitura do *bit* F:

Se $f_i = f_j$ então retorna 1;
senão retorna 0;

Escrita do *bit* F (para o valor 1):

Se $f_i \neq f_j$ então f_i := not f_i;

Como será demonstrado ao final deste capítulo, a variável w_i (w_j) deve ser consistente, ou seja, se um processo k lê o novo valor da variável w_i, o processo i, lendo esta variável após a leitura feita por k, deverá também obter o novo valor (a menos que seja iniciada uma escrita nesta variável).

Para isto, a variável w_i (w_j) deve ser implementada como duas variáveis, wp_i e wq_i (wp_j e wq_j), e sua operação de escrita definida da

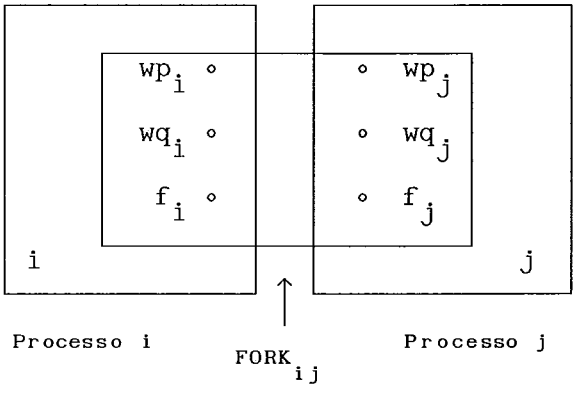
seguinte forma:

wp_i = verdadeiro (wp_j = verdadeiro)

wq_i = verdadeiro (wq_j = verdadeiro)

A operação de leitura de w_i é a leitura de wp_i pelo processo i e de wq_i pelos outros processos no sistema ({1..n} - {i}).

Assim, cada processo necessita de 3 *bits* (wp, wq e f) para cada processo com quem se comunica. A estrutura de dados FORK é composta de 6 *bits safe* por par de processos, como mostra a figura abaixo:



Para simplificar a apresentação foi criado o tipo abstrato de dados chamado **ARROW**. O ARROW pode ser considerado como um FORK em que os estados $OFERECIDO_i$ e EM_USO_j foram unidos para formar o estado $i \longrightarrow j$ e os estados $OFERECIDO_j$ e EM_USO_i unidos para formar o estado $j \longrightarrow i$. Da mesma forma, as operações $redirect_i$ e $redirect_j$ são consideradas como sendo $take_i$ seguido de $offer_i$ e $take_j$ seguido de $offer_j$, respectivamente. A operação $read_k(ARROW_{ij})$ para um processo k é definida como:

```
caso read_k(FORK_ij) seja
     OFERECIDO_j ou EM_USO_i : retorna j ⟶ i;
     OFERECIDO_i ou EM_USO_j : retorna i ⟶ j;
fim
```

e um $redirect_i(ARROW_{ij})$ como uma primitiva $take_i(FORK_{ij})$ seguida por uma $offer_i(FORK_{ij})$.

III.6 O ALGORITMO

O algoritmo que será apresentado foi inicialmente concebido para solucionar o problema de Exclusão-ℓ através do uso de registradores *safe*. Entretanto, grande parte do mesmo é independente do tipo de registrador utilizado, podendo ser simplificado para ser usado com outros tipos de registradores **SWMR** mais poderosos. Esta simplificação deve ser feita nas estruturas de dados de comunicação e em suas primitivas de manipulação (*take* e *offer*).

Para obter a Exclusão-ℓ foi criada uma estrutura de dados chamada GRAPH que consiste de uma instância do tipo $ARROW_{ij}$ (denominada G_ARROW_{ij}) entre cada par de processos i e j $\in$ {1..n}. Além disto, cada processo k mantém um **SWMR** *safe bit* x_k que é atribuído verdadeiro ao deixar a seção Restante e falso ao retornar a ela. A união de todos x_k, k $\in$ {1..n} é chamada de X. Todo processo que deseja executar a sua seção crítica deve ler X e GRAPH. Ao ler GRAPH o processo i obtém um grafo G(i) com n nós, sendo que a direção da conexão entre os vértices i e j é dada pelo estado de G_ARROW_{ij}.

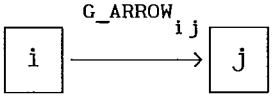


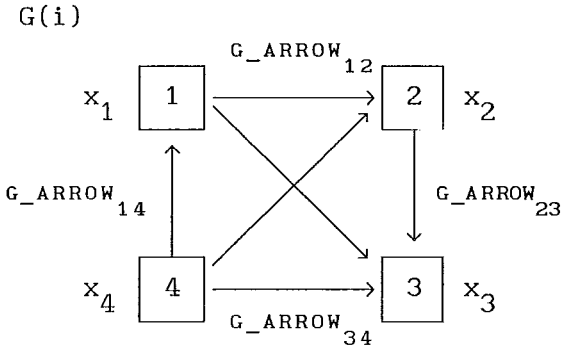
Figura III.1 - Estrutura de dados compartilhada pelos processos i e j

É importante observar que o grafo G(i) pode nunca ter existido, uma vez que a leitura de um único arco (j,k) envolve a leitura de seis *bits* da estrutura G_ARROW_{jk} concorrentemente com possíveis

redirecionamentos feitos por j e k.

Seja $R_i(G)$ o conjunto de todos os nós alcançáveis através de um caminho direcionado iniciando no nó i do grafo direcionado G(i) (incluindo i). Este conjunto é chamado de conjunto de alcançabilidade.

Ex: Grafo G(i) gerado pela leitura de X e GRAPH feita pelo processo i



GRAPH

$G\_ARROW_{12} = 1 \rightarrow 2$

$G\_ARROW_{13} = 1 \rightarrow 3$

$G\_ARROW_{14} = 4 \rightarrow 1$

$G\_ARROW_{23} = 2 \rightarrow 3$

$G\_ARROW_{24} = 4 \rightarrow 2$

$G\_ARROW_{34} = 4 \rightarrow 3$

$X = \{ x_1, x_2, x_3, x_4 \}$

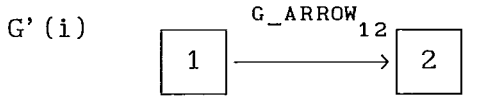
Figura III.2 - Exemplo dos grafos G(i) e GRAPH

No caso de i = 1 temos $R_1(G) = \{1,2,3\}$

É definida uma rotina $\Re$(i,GRAPH,X) que executa o seguinte procedimento:

1- Lê X e GRAPH
2- Obtém o grafo G'(i) (sub-grafo de G(i)) constituído de todos os nós k tais que a leitura de x_k por i retorna verdadeiro (processos fora da seção Restante)
3- Retorna $R_i(G'(i))$

Se no exemplo da figura III.2 tivessemos x_1 e x_2 verdadeiros e x_3 e x_4 falsos o subgrafo G'(i) teria a seguinte forma:



x_1 = verdadeiro x_2 = verdadeiro

Figura III.3

Assim temos $\Re(i,GRAPH,X) = R_i(G'(i)) = \{1,2\}$

Se a cardinalidade do conjunto de alcançabilidade retornado por $\Re(i,GRAPH,X)$ for menor ou igual a ℓ, o nó pode entrar em sua seção crítica. A razão para se escolher a alcançabilidade foi a necessidade de se ter uma relação transitiva. A transitividade assegura que em um grupo de ℓ + 1 processos, algum processo terá todos os outros à sua frente.

Para todo processo i $\in$ {1..n}, o algoritmo abaixo satisfaz a Exclusão-ℓ:

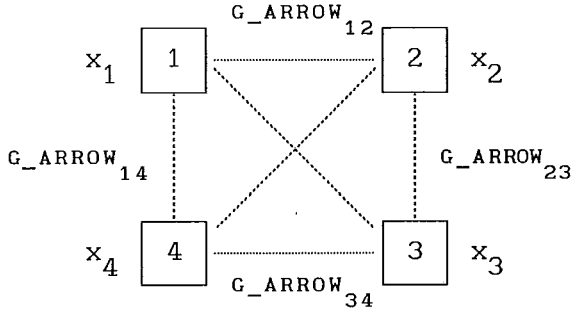
```
faça para sempre
   Restante;
   x_i = verdadeiro;
   para todo j ∈ {1..n} faça
        redirect(i,j,GRAPH);
L: oracle(GRAPH);
   Se | ℜ(i,GRAPH,X) | > ℓ então vá para L;
   seção crítica;
   x_i = falso;
fim do faça
```

onde redirect(i,j,GRAPH) é a execução de $redirect_i(G\_ARROW_{ij})$ e oracle(GRAPH) é uma rotina que será substituída por um mecanismo de prevenção de *deadlock*. Esta rotina, quando chamada por i, escolhe um subconjunto arbitrário de ARROWs no qual i é um alterador e as redireciona.

O funcionamento deste algoritmo pode ser melhor compreendido através do exemplo a seguir:

• Seja um sistema composto de quatro processos, sendo que no máximo dois processos podem executar simultaneamente suas seções críticas (n = 4 , ℓ = 2). Inicialmente, todos os processos do sistema estão fora das suas seções críticas. O grafo G(i), observado pelo processo i é mostrado abaixo:

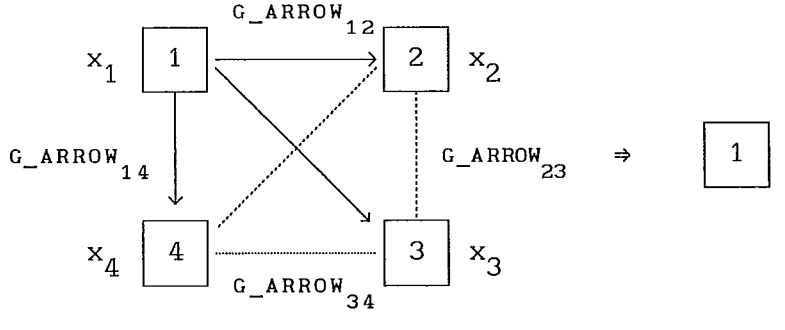
G(i)



X = { F, F, F, F } onde, F = falso
V = verdadeiro

1) O processo 1 tenta entrar na sua seção crítica (Faz x_1 = verdadeiro)

G(1) G'(1)

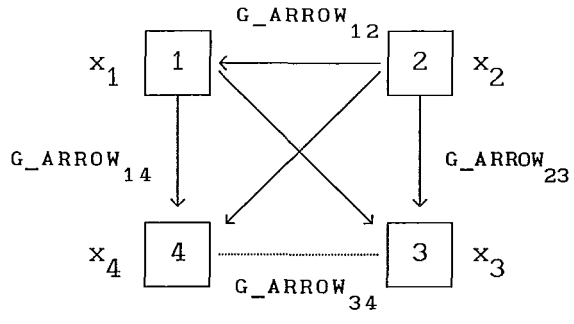


X = { V, F, F, F } $\Re(1,GRAPH,X)$ = { 1 }

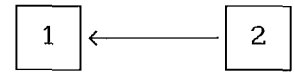
De acordo com o algoritmo, como $|\ \Re(1,GRAPH,X)| < 2$, o processo 1 entrará em sua seção crítica.

2) O processo 2 tenta entrar na sua seção crítica (Faz x_2= verdadeiro)

G(2)



G'(2)



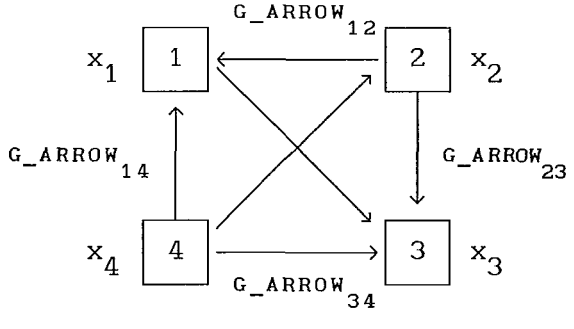
X = { V, V, F, F }

$\Re(2,GRAPH,X)$ = { 1,2 }

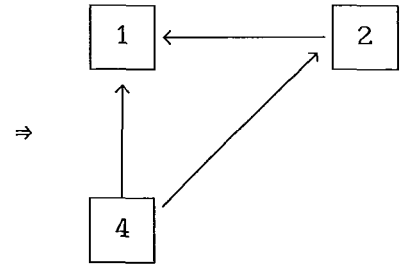
Como $|\ \Re(2,GRAPH,X)\ |$ = 2 , o processo 2 entrará em sua seção crítica

3) O processo 4 tenta entrar na sua seção crítica (Faz x_4= verdadeiro)

G(4)



G'(4)



X = { V, V, F, V }

$\Re(4,GRAPH,X)$ = { 1,2,4 }

Como $|\ \Re(4,GRAPH,X)\ |$ > 2 , o processo 4 não poderá entrar em sua seção crítica, devendo repetir este procedimento até que um dos processos deixe a sua seção crítica (fazendo x_k= falso).

III.7 AUSÊNCIA DE DEADLOCK

Esta construção pode apresentar *deadlock*, uma vez que muitos

processos podem constantemente ter sua alcançabilidade maior que ℓ, nunca entrando na sua seção crítica. Para evitar este problema, a *procedure* oracle(GRAPH) deve redirecionar os ARROWS na direção dos processos que têm maiores *ids* (número de identificação).

Se não houver falha nos processos o *deadlock* será evitado, pois em um grupo de processos bloqueados, aquele que possuir o *id* mais alto, eventualmente terá todas as setas (ARROWs) redirecionadas para ele, tornando sua alcançabilidade menor que ℓ. Infelizmente, isto não continua sendo verdadeiro caso haja uma falha em um único processo. Este tipo de construção não é justa pois dá prioridade aos processos que possuem números de identificação maiores. A questão de justiça será abordada na próxima subseção, onde serão introduzidos os *ids* dinâmicos.

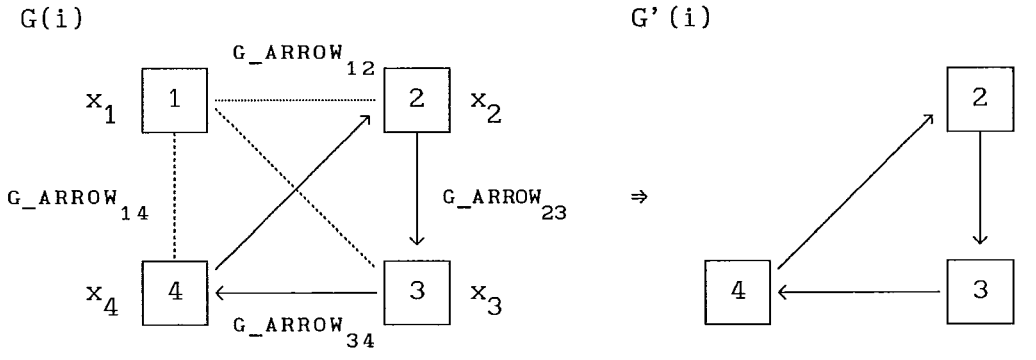
Como mencionado anteriormente, a solução original do problema de Exclusão-ℓ apresentada por DOLEV, GAFNI e SHAVIT (em [1]) tem como objetivo satisfazer as propriedades de ausência de *deadlock* e de *lockout* mesmo em caso de falha dos processos. Como esta tese se propõe basicamente a comparar o desempenho das soluções com os três tipos de registradores **SWMR**, assumimos que os processos não falham durante a execução do sistema. Desta forma, o algoritmo pôde ser simplificado removendo-se as rotinas que tratam especificamente dos casos em que há falhas nos processos.

O exemplo a seguir demonstra a ocorrência de *deadlock* e o procedimento utilizado para evitá-lo.

• Assumimos, neste exemplo, que inicialmente todos os processos estão fora de suas seções críticas. Suponhamos que os processos 2, 3 e 4 decidam entrar em suas seções críticas ao mesmo tempo. Estes três processos irão concorrentemente executar x_i= verdadeiro e redirecionar suas estruturas ARROW para os outros processos. Este redirecionamento

simultâneo das estruturas pode levar à formação de ciclos no grafo G'(i) e conseqüentemente deixar o sistema em *deadlock*. Neste caso, os processos teriam continuamente suas alcançabilidades maiores que ℓ, permanecendo para sempre em *loop* e desta forma nunca executando suas seções críticas.

Ex: Observe que os processos 2, 3 e 4 possuem alcançabilidade > 2 devido ao ciclo no grafo G'(i).

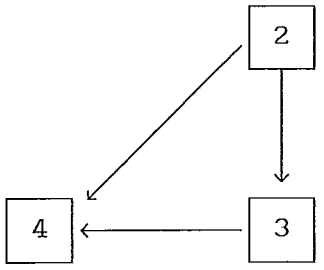


X = { F, V, V, V }

$\Re$(i,GRAPH,X) = { 2,3,4 }

Para remover este *deadlock*, a rotina oracle(), executada pelos processos dentro do *loop* na seção de entrada, faz com que os processos redirecionem as suas estruturas ARROW na direção dos processos que possuem *ids* maiores. Neste exemplo, o processo 2 irá redirecionar a seta na direção do processo 4, quebrando, desta forma, o ciclo existente no grafo G'(i). O grafo G'(i), obtido após a execução da rotina oracle(), é mostrado a seguir:

G'(i)



$\Re(2,GRAPH,X) = \{ 2,3,4 \}$

$\Re(3,GRAPH,X) = \{ 3,4 \}$

$\Re(4,GRAPH,X) = \{ 4 \}$

No grafo acima, o processos 3 e 4 possuem alcançabilidades 2 e 1 respectivamente, podendo assim, entrar em suas seções críticas.

III.8 AUSÊNCIA DE LOCKOUT

A rotina oracle(), utilizada para evitar *deadlocks*, favorece os processos que possuem *ids* maiores e, por esta razão, introduz a possibilidade de *lockout*, situação em que os processos com *ids* menores ficam retidos em suas seções de entrada, nunca conseguindo assim executar as suas seções críticas.

Para evitar o *lockout* deve ser introduzido um mecanismo para a criação de *ids* que são dinamicamente incrementados. Desta forma, os *ids* dos processos bloqueados em *lockout* são incrementados até que se tornem maiores do que o dos outros processos que não estão em *lockout*. Para criar *ids* dinâmicos deve-se utilizar uma nova estrutura de dados chamada ID, que consiste de uma coleção de estruturas do tipo $FORK_{ij}$ (denominadas ID_FORK_{ij}), cada uma pertencendo a um par de processos i,j $\in \{1..n\}$, de maneira similar ao GRAPH.

Cada processo que desejar entrar na sua seção crítica tentará, continuamente, obter todos os FORKS oferecidos a ele. O *id* dinâmico será igual ao número de FORKS (no estado EM_USO) que o processo possui. O processo só oferecerá os FORKS que possui depois de deixar a seção crítica. Assim um processo que está bloqueado e continuamente coletando

FORKS terá seu *id* incrementado gradualmente. A figura III.3 fornece um exemplo de *id* dinâmico.

Estrutura ID

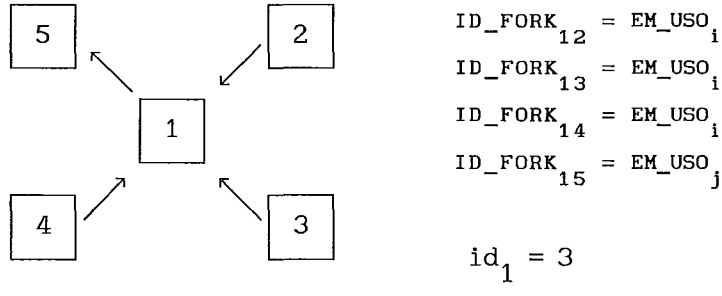


Figura III.3 - Exemplo de ID dinâmico

Na figura acima, o *id* do processo 1 é igual ao número de *forks* que possui no estado EM_USO. Neste caso $id_1 = 3$.

Foram criadas 3 rotinas para implementar o *id* dinâmico. São elas:

• Increment_your (ID) - Tenta obter para o processo todos os *forks* da estrutura ID oferecidos a ele. É chamada na seção de entrada do processo.

• Initialize_your (ID) - Oferece todos os *forks* da estrutura ID que o processo possui. É chamada após a execução da seção crítica.

• Observe (ID) - Obtém o número de *forks* da estrutura ID que os processos possuem (no estado EM_USO) e atualiza seus *ids*. A rotina oracle(GRAPH) utiliza posteriormente estes *ids* dinâmicos para remover de forma justa os eventuais *deadlocks* sem provocar *lockout*.

O algoritmo abaixo exemplifica o procedimento para a obtenção de *ids* dinânicos:

faça para sempre

```
L: increment_your(ID);
   observe(ID);
   oracle(GRAPH);
   Se testa() = verdadeiro então vá para L;
   seção crítica;
   initialize_your(ID);
fim do faça;
```

A rotina testa() definida neste exemplo, retorna arbitrariamente verdadeiro ou falso, simulando a entrada ou não do processo em sua seção crítica. No algoritmo acima, os processos que são impedidos de entrar em suas seções críticas, retornam ao localizador L e tentam aumentar o seu *id* chamando a rotina increment_your(ID). Como os processos com *ids* maiores têm prioridade, o aumento do *id* fará com que o processo eventualmente entre em sua seção crítica. Os *forks* só são liberados após a execução da seção crítica com a chamada da rotina initialize_your(ID).

O algoritmo abaixo resolve o problema de Exclusão-ℓ satisfazendo as propriedades de ausência de *deadlock* e de *lockout*, caso não haja falha nos processos.

```
faça para sempre
   Restante;
   x_i = verdadeiro;
   para todo j ∈ {1..n} faça
        redirect(i,j,GRAPH);
L: increment_your(ID);
   observe(ID);
   oracle(GRAPH);
   Se | ℜ(i,GRAPH,X) | > ℓ então vá para L;
   seção crítica;
   inicialize_your(ID);
   x_i = falso;
```

```
fim do faça;
```

onde:

```
increment_your(ID):
   para todo j ∈ {1..n} faça
        take(i,j,ID);

initialize_your(ID):
   para todo j ∈ {1..n} faça
        offer(i,j,ID);

observe(ID):
   para todo j em {1..n} faça
       contador := 0;
       para todo k em {1..n} faça
          se read(j,k,ID) = EM_USO_j então
              contador := contador + 1;
       id_j = (contador,j);

oracle(GRAPH):
   para todo j ∈ {1..n} faça
     se ((id_j > id_i) ou (não x_j)) então redirect(i,j,GRAPH);
        senão se (id_j = id_i e j > i) então redirect(i,j,GRAPH);
```

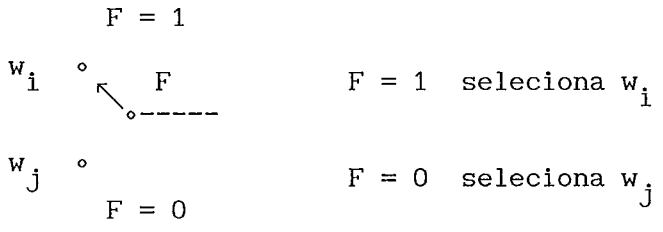
III.9 FUNCIONAMENTO DO FORK E DE SUAS PRIMITIVAS

A função da estrutura abstrata de dados FORK, é fornecer um meio de serializar operações concorrentes em registradores *safe*, de modo a substituir a atomicidade na sincronização de processos. Esta serialização é obtida através da mudança ordenada dos estados da estrutura, como mostrado na seção III.5.

Como foi descrito no capítulo II, uma leitura concorrente com uma escrita em um registrador *safe*, pode retornar qualquer um dos valores permitidos para o registrador. Entretanto, esta mesma operação

concorrente em uma estrutura FORK, constituída de registradores *safe*, retorna apenas o valor antigo ou o novo, de maneira semelhante a um registrador atômico. Vejamos como isto é obtido:

O bit F funciona como uma chave que seleciona o processo que determina o estado do FORK. Podemos observer isto na figura abaixo:



Quando F = 1, a variável w_i (pertencente ao processo i) é selecionada e determina o estado do FORK.

w_i = falso ⇒ estado OFERECIDO$_i$

w_i = verdadeiro ⇒ estado EM_USO$_i$

Quando F = 0, é a variável w_j que determina o estado do FORK.

w_j = falso ⇒ estado OFERECIDO$_j$

w_j = verdadeiro ⇒ estado EM_USO$_j$

Assim, por exemplo, quando F = 0, o processo i pode escrever um novo valor em w_i sem alterar o estado do FORK. É desta forma que se realizam as transições do FORK. Por exemplo, para alterar o estado de OFERECIDO$_j$ para EM_USO$_i$, as variáveis w_i e F, devem ser escritas na seguinte ordem:

w_i := verdadeiro (observe que o estado não é alterado, pois F = 0)

F := 1 (Realiza a transição de estado)

No exemplo acima, o estado só é alterado quando se inicia a atribuição ao *bit* F. Uma leitura do FORK, executada simultaneamente com uma escrita neste *bit*, irá retornar $OFERECIDO_J$ ($F = 0 \wedge w_j$ = falso) ou EM_USO_i ($F = 1 \wedge w_i$ = verdadeiro), uma vez que a leitura concorrente do *bit* F só pode retornar 0 ou 1.

Como foi mencionado anteriormente neste capítulo, a ordem de leitura dos *bits* F, w_j e w_i é fundamental para o funcionamento correto da estrutura.

A transição entre os estados EM_USO_i e $OFERECIDO_i$ é feita atribuindo-se falso a w_i (ver primitiva $offer_i$). Igualmente, neste caso, uma leitura do FORK concorrente com a escrita de w_i, irá obter o estado antigo ou o novo.

O algoritmo de Exclusão-ℓ descrito neste capítulo, requer que o o FORK satisfaça a seguinte propriedade:

- Seja R1 uma leitura pelo processo k e R2 uma leitura pelo alterador j que segue estritamente R1. Se R1 retorna $OFERECIDO_i$ e nenhuma alteração de estado é feita pelo processo j no intervalo [R1..R2], então R2 deve retornar o mesmo resultado.

Esta propriedade implica que, se um processo lê $OFERECIDO_i$, então o alterador j também deverá ler este mesmo estado. Para satisfazer este requerimento, a variável w_i (w_j) é implementada por duas variáveis, wq_i e wp_i (wq_j e wp_j), e sua operação de escrita realizada da seguinte forma:

wp_i := verdadeiro

wq_i := verdadeiro

A leitura de w_i é a de wp_i pelo processo j e a de wq_i por todos os outros processos. Desta forma, a propriedade acima é satisfeita, uma vez que a variável lida por j é sempre atualizada antes da outra.

É importante observarmos, que as mudanças de estado do FORK ocorrem necessariamente em uma determinada ordem e apenas se certas condições forem satisfeitas. Estas condições, chamadas de pré-condições, representam o estado corrente do FORK. Por exemplo, a pré-condição da primitiva $take_i$ é $\{F = 0 \wedge w_j = falso\}$, ou seja, o estado $OFERECIDO_j$. Uma condição do tipo $\{F = 0 \wedge w_j = verdadeiro\}$, significa que a leitura do FORK, por qualquer processo, retorna F = 0 e w_j = verdadeiro. A pós-condição representa o estado em que o FORK é deixado ao final da execução da primitiva.

Assim, as primitivas *take* e *offer* só modificam o estado do FORK, se ao serem executadas, o FORK estiver em determinado estado.

A seguir são descritas as primitivas *take* e *offer*, juntamente com suas pré-condições e pós-condições.

$take_i$:

$\{F = 0 \wedge w_j = falso\}$ /* Pré-condição */

Se read($FORK_{ij}$) = $OFERECIDO_j$ então

w_i := verdadeiro;

$\{F = 0 \wedge w_i = verdadeiro\}$

F := 1;

$\{F = 1 \wedge w_i = verdadeiro\}$ /* Pós-condição */

offer$_i$:

{F = 1 $\wedge$ w_i = verdadeiro}

Se read($FORK_{ij}$) = EM_USO_i então

w_i := falso;

{F = 1 $\wedge$ w_i = falso}

take$_j$:

{F = 1 $\wedge$ w_i = falso}

Se read($FORK_{ij}$) = $OFERECIDO_i$ então

w_j := verdadeiro;

{F = 1 $\wedge$ w_j = verdadeiro}

F := 0;

{F = 0 $\wedge$ w_j = verdadeiro}

offer$_j$:

{F = 0 $\wedge$ w_j = verdadeiro}

Se read($FORK_{ij}$) = EM_USO_j então

w_j := falso;

{F = 0 $\wedge$ w_j = falso}

III.9.1 PROVA INFORMAL DE VALIDADE

Dada uma instância da estrutura FORK, $FORK_{ij}$, suponhamos que inicialmente temos $w_i = w_j$ = falso e F = 0.

Com estas condições iniciais, a única primitiva que tem sua pré-condição satisfeita é a $take_i$. As pré-condições das outras primitivas, que podem ser executadas concorrentemente com $take_i$, só serão satisfeitas após o término de $take_i$, e por esta razão, a pré-condição de $take_i$ permanece inalterada até que esta conclua sua

operação.

Ao final da execução de $take_i$, a pré-condição de $offer_i$, {F = 1 $\wedge$ w_i = verdadeiro}, é satisfeita. Até a execução da atribuição a w_i em $offer_i$, não é satisfeita a pré-condição de nenhuma outra primitiva. Unicamente após o início desta atribuição é que a pré-condição de $take_j$, e apenas de $take_j$, é satisfeita. Assim, i pode estar escrevendo em w_i enquanto j executa um $take_j$ e, ainda assim, a pós-condição de $take_j$ continua sendo válida, dado que sua pré-condição era verdadeira.

A pré-condição para qualquer alteração provocada por i, não é satisfeita até que seja feita a atribuição de w_j pela primitiva $offer_j$. Isto só poderá ocorrer depois que a execução da primitiva $take_j$ se completar, quando será satisfeita a pré-condição para $offer_j$.

Podemos concluir através dos argumentos acima que, se a pré-condição de uma primitiva é satisfeita, esta continuará válida até que ocorra uma mudança de estado e, desde que esta não ocorra, as pré-condições das outras primitivas não são satisfeitas.

CAPÍTULO IV

MÉTODOS DE AVALIAÇÃO

IV.1 IMPLEMENTAÇÃO

O algoritmo proposto em [1] tem como objetivo solucionar o problema de Exclusão-ℓ através do uso dos registradores mais fracos, ou seja, utilizando registradores *safe* e dispensando desta forma a obtenção da atomicidade.

Nesta tese o algoritmo foi adaptado para ser utilizado com os outros tipos de registradores **SWMR**. Esta adaptação permitiu que fossem comparados os resultados práticos de desempenho do sistema em relação ao tipo de registrador utilizado. Foram implementadas três versões para o programa, sendo uma para cada tipo de registrador.

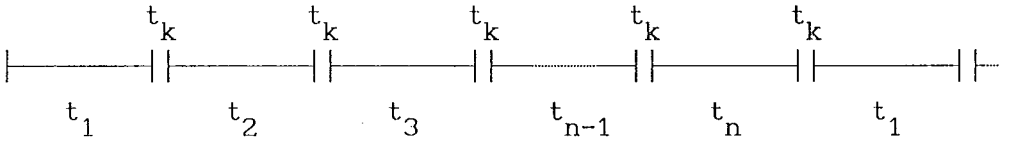
O algoritmo foi desenvolvido nas linguagens de programação C e *Assembly* 8086. As medidas experimentais foram obtidas executando o programa em um microcomputador DELL System 310, compatível com o padrão IBM PC AT, com microprocessador 80386 e Sistema Operacional MS-DOS 3.3.

Os registradores **SWMR** foram implementados por rotinas que simulam o comportamento das suas operações de leitura e escrita, utilizando as células de memória atômica disponíveis no equipamento utilizado.

Para simular a execução de processos concorrentes em um único processador, foi necessário implementar um *kernel* preemptivo, com algoritmo de escalonamento do tipo *round-robin* (circular), de forma a chavear a execução entre os processos. A fatia de tempo (*time-slice)* de cada processo foi escolhido muito menor que o tempo de execução de uma operação (simulada) de escrita (e de leitura) dos registradores que constituem as estrutura de dados de sincronização. Quanto menor for a

fatia de tempo de cada processo, maior será a chance de ocorrerem operações simultâneas de leitura e escrita e, também, mais próximo o sistema estará da situação real que desejamos simular, em que existe um processador para cada processo. Por outro lado, a fatia de tempo não pode ser diminuída indefinidamente, pois como o tempo de execução do *kernel* (t_k) é fixo e este é executado ao final de cada fatia de tempo para troca de contexto, o tempo total (duração do teste) se tornaria demasiadamente grande.

Assim, os processos são executados de forma cíclica, ou seja, ao final da execução do último processo, é reiniciada a execução do primeiro processo. A figura abaixo mostra a divisão do tempo de processamento entre os *n* processos do sistema.



onde,

t_k = tempo de execução do kernel para troca de contexto

e t_n = fatia de tempo de processador alocada para o processo n

A fila de processos prontos foi feita circular e composta de BCPs (Bloco de Controle do Processo), como mostrado na figura IV.1. Cada processo está associado a um BCP, que contém um ponteiro para o BCP do próximo processo a ser executado, além de uma área reservada para salvar o contexto do processo.

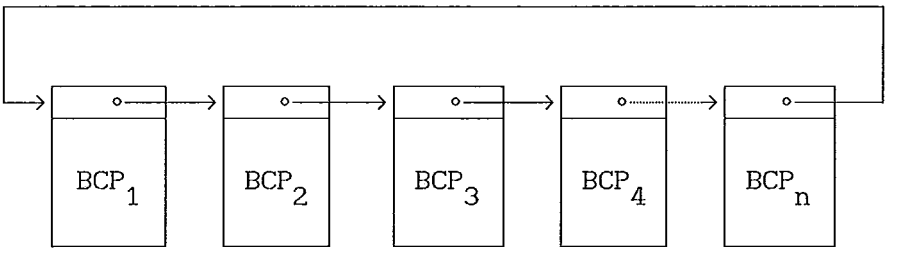


Figura IV.1 – Fila de processos prontos

O tempo de execução de cada processo foi medido através de um contador de tempo real disponível no equipamento utilizado. Este temporizador é programado pelo *kernel* para gerar um pulso de duração igual à fatia de tempo especificada. Ao final da fatia de tempo este pulso interrompe a execução do processo corrente e transfere o controle de volta para o *kernel* que salva o contexto e seleciona o próximo processo a ser executado.

A comunicação entre os processos é feita através de uma memória compartilhada, pertencente ao *hardware* utilizado, onde foram implementadas as estruturas de dados e simulados os registradores **SWMR**.

O programa que implementa o algoritmo é composto de uma parte dependente e outra independente do tipo de registrador **SWMR** utilizado. A parte independente é responsável pela implementação das propriedades de Exclusão-ℓ, ausência de *deadlock* e ausência de *lockout*. Nesta parte são manipuladas as estruturas de dados (grafos) que estabelecem a ordem de precedência entre os processos.

A parte dependente contém as primitivas que manipulam as estruturas abstratas de comunicação, ou seja, os **FORKS** e **ARROWS**. Existe um módulo deste para cada tipo de registrador.

IV.2 MÉTODO DE OBTENÇÃO DOS RESULTADOS

O problema de Exclusão-ℓ apresenta um número de variáveis

bastante elevado, o que torna complexa uma análise comparativa muito ampla. Por esta razão, decidimos restringir o conjunto das variáveis, fixando algumas delas, e utilizando apenas as mais relevantes e de maior influência no desempenho do sistema.

O programa desenvolvido permite a alteração dos seguintes parâmetros:

1) n = Número total de processos no sistema
2) ℓ = Número máximo de processos na seção crítica
3) T_c = Tempo de execução da seção crítica
4) T_r = Tempo de execução da seção Restante
5) Tipo de registrador

Fizemos também com que todos os processos do sistema se comportassem de forma idêntica, possuindo os mesmos valores para T_r e T_c e a mesma velocidade de processamento (determinada pelo *hardware* utilizado).

Ao final da execução do programa, obtemos uma medida de eficiência que é dada pelo número médio de vezes que os processos executam suas seções críticas por unidade de tempo de execução.

Para permitir uma análise comparativa da eficiência das três soluções implementadas, as rotinas que simulam os registradores foram ajustadas de forma que os tempos de acesso das operações de leitura e escrita não-concorrentes fossem iguais. Em caso de operações concorrentes no mesmo registrador, a rotina que simula o registrador atômico: força que o processo que requisitou a operação espere pelo término da operação corrente, gera um atraso correspondente ao tempo de chaveamento interno da memória e executa a operação requisitada. Esta rotina simula o comportamento de memórias do tipo *multi-porta* descrito anteriormente no capítulo II.

O uso de registradores *safe* e regulares elimina a necessidade de

contenção no barramento, pois o acesso pode ser feito concorrentemente em barramentos separados. Entretanto, os algoritmos que se baseiam nestes registradores são mais complexos e conseqüentemente necessitam de um tempo de processamento maior para efetuarem o controle de concorrência.

Para analisar o comportamento da solução implementada, foram feitas medidas variando-se os parâmetros descritos anteriormente, da seguinte forma:

1o caso) Mantendo n fixo e variando ℓ, de modo a que a razão ℓ/n varie de 0,1 a 1. Este caso visa a análise do comportamento do sistema com a variação do número de recursos. É esperado que o desempenho do sistema aumente com o aumento do número de recursos disponíveis (ℓ). Para a obtenção das medidas experimentais, n foi mantido fixo em 20 processos enquanto que ℓ variou de 2 a 20 recursos.

2o caso) Mantendo ℓ fixo e variando n, de modo que a razão ℓ/n varie de 0,1 a 1. Este caso visa a análise do comportamento do sistema com a variação do número de processos disputando o acesso a um número fixo de recursos. É esperado que o aumento do número de processos (n) aumente a disputa pelos recursos, reduzindo assim o desempenho do sistema. Para a obtenção das medidas experimentais, ℓ foi mantido fixo em 2 recursos enquanto que n variou de 2 a 20 processos.

Para cada um dos casos acima foram feitas medidas variando-se os parâmetros que envolvem os tempos de execução das seções dos processos T_r e T_c. Quatro casos foram analisados, considerando-se duas possibilidades extremas de tempo (P e G) para T_r e T_c.

T_r	T_c
P	P
P	G
G	P
G	G

P = tempo de execução muito pequeno

G = tempo de execução muito grande

Todas estas medidas foram obtidas para cada tipo de registrador (SAFE, REGULAR e ATÔMICO).

O procedimento para a obtenção das medidas descritas acima foi repetido 10 vezes para eliminar possíveis erros e discrepâncias, sendo a média destas medidas o resultado final adotado. Os resultados obtidos foram posteriormente apresentados em forma de gráficos para facilitar a sua visualização e análise comparativa. Como foi mencionado anteriormente a medida adotada para análise de desempenho da solução foi o número médio de vezes que cada processo do sistema executa a sua seção crítica dividido pelo seu tempo de execução. Os gráficos descrevem a variação deste resultado de desempenho (eixo y) de acordo com a variação da relação ℓ/n (eixo x).

IV.3 ADAPTAÇÃO DO ALGORITMO

O algoritmo proposto por [1] foi inicialmente criado visando solucionar o problema de Exclusão-ℓ de forma justa, diretamente através de registradores *safe*, eliminando a etapa intermediária de construção de registradores atômicos. Novas estruturas de dados foram idealizadas de modo a substituir a atomicidade e a permitir a sincronização dos processos sem impor nenhum tipo de espera aos processos. Como descrito no capítulo III, as primitivas de sincronização *take* e *offer* impõem uma serialização das operações sobre as estruturas de dados (FORK e ARROW) e eliminam a espera por parte dos processos concorrentes.

Entretanto, o algoritmo proposto é geral e pode ser simplificado no caso de utilização de registradores mais poderosos, como os atômicos. Para isto devem ser alteradas as primitivas que manipulam as estruturas de dados, de modo a satisfazerem requerimentos mais fracos.

Desta forma, no caso de registradores atômicos, as primitivas *take* e *offer* foram simplificadas de modo a manipularem apenas 2 *bits* atômicos, em vez dos 6 *bits safe* (ou regulares) da estrutura de dados FORK. Estes 2 *bits* representam os 4 estados possíveis para uma estrutura de dados do tipo FORK (EM_USO_i, EM_USO_j, $OFERECIDO_i$, $OFERECIDO_j$). A operação de leitura desta estrutura também foi simplificada, como mostra o exemplo a seguir:

A rotina $read_k(FORK_{ij})$ para registradores *safe* e regulares é definida da seguinte forma:

```
read_k(FORK_ij)
{
  Se (f_i = f_j) então {                              /* F = 1 */
                Se (wq_i) então retorna(EM_USO_i);
                          senão retorna(OFERECIDO_i);
              }
            senão {                                   /* F = 0 */
                Se (wq_j) então retorna(EM_USO_j);
                          senão retorna(OFERECIDO_j);
            }
}
```

Esta mesma rotina, no caso de registradores atômicos, lê diretamente o conteúdo do registrador.

```
read_k(fork_ij)
{
  retorna(fork_ij)
}
```

onde $fork_{ij}$ é um registrador atômico de 2 *bits*.

A primitiva $take_j$, para registradores *safe* e regular, é mostrada abaixo:

```
take_j(fork_ij)
{
  Se ( read_k(fork_ij) = OFERECIDO_i ) então {
      wp_j := verdadeiro;                    /* w_j = verdadeiro */
      wq_j := verdadeiro;
      Se (f_i = f_j) então  f_j := not f_j;   /* F = 0 */
  }

}
```

A mesma primitiva $take_j$, para registradores atômicos, é escrita da seguinte forma:

```
take_j(fork_ij)
{
  Se (fork_ij = OFERECIDO_i) então fork_ij := EM_USO_j;
}
```

onde $fork_{ij}$ é um registrador atômico de 2 *bits*.

Podemos observar no exemplo acima, que o número de operações de leitura e escrita é bem menor nas primitivas para registradores atômicos. Isto ocorre porque, sendo os registradores atômicos mais poderosos, a estrutura FORK se torna bem mais simples, podendo ser implementada com um número menor de *bits*.

IV.4 SIMULAÇÃO DOS REGISTRADORES

Como o equipamento utilizado possui apenas memórias atômicas com células de 8 *bits*, foi necessário criar um método para simular o comportamento dos registradores booleanos *safe*, regulares e atômicos

necessários para as implementações da solução.

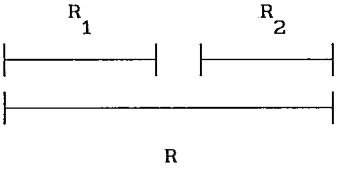
Nos três casos, a simulação foi implementada através da criação das rotinas $read_k$ e $write_k$, que têm a função de ler e escrever um *bit* de informação nos registradores.

IV.4.1 SIMULAÇÃO DE REGISTRADORES REGULARES

Seja R uma operação de leitura em um registrador booleano regular. A operação R pode ser implementada através das operações de leitura atômicas consecutivas R1 e R2, da seguinte forma:

Rotina $read_k()$

- Lê o *bit* do registrador atômico (R_1)
- Lê novamente o mesmo *bit* do registrador atômico (R_2)
- Retorna aleatoriamente R_1 ou R_2



R_1 e R_2 são operações de leituras atômicas
R é uma operação de leitura em um registrador regular

Figura IV.2 - Simulação de registradores regulares

Se R1 ≠ R2 podemos assumir que a leitura R foi executada concorrentemente com uma operação de escrita que modificou o valor armazenado no registrador.

Se a operação de leitura for concorrente com uma operação de escrita que não altere o valor armazenado, esta deve retornar o valor correto (o último valor escrito), pois o registrador regular sempre

retorna o valor antigo ou o novo.

Como o registrador é booleano, este só pode assumir os valores 0 e 1.

O tempo de acesso do registrador regular simulado é igual ao intervalo correspondente ao início da leitura R_1 até o final da leitura R_2.

IV.4.2 SIMULAÇÃO DE REGISTRADORES SAFE

A simulação de registradores *safe* booleanos é um pouco mais complexa. É necessário criar um método para detectar se a operação de leitura está sendo executada simultaneamente com uma operação de escrita, pois mesmo no caso de uma escrita concorrente que não altere o valor armazenado, a leitura deve retornar qualquer um dos valores possíveis para o registrador (neste caso 0 ou 1).

Para detectar a concorrência é utilizado um *bit* a mais por registrador. Este *bit*, denominado *bit* de concorrência, é ligado pela rotina $write_k$ toda vez que for executada uma operação de escrita no registrador *safe*. O procedimento de detecção de operações concorrentes é realizado pela rotina $read_k$, da forma descrita a seguir:

Seja R uma operação de leitura em um registrador booleano *safe*. A operação R pode ser implementada através das operações de leitura atômicas consecutivas R1 e R2, da seguinte forma:

Rotina $read_k()$

- Desabilita interrupções
- Lê o *bit* do registrador atômico (R_1)
- Desliga o *bit* de concorrência
- Habilita interrupções
- Simula o tempo de acesso da memória

- Desabilita interrupções
- Lê novamente o mesmo *bit* do registrador atômico (R_2)
- Habilita interrupções
- Se o *bit* de concorrência estiver desligado retorna R_1 (ou R_2) senão retorna 0 ou 1 aleatoriamente

Se a leitura R for concorrente com uma escrita no mesmo registrador, a rotina $read_k$ irá detectar a concorrência através do *bit* de concorrência ativado e retornará aleatoriamente um dos valores possíveis para o registrador (neste caso 0 ou 1).

IV.4.3 SIMULAÇÃO DE REGISTRADORES ATÔMICOS

Também foi necessário simular o comportamento dos registradores atômicos booleanos a partir das células de memória atômicas de 8 *bits* disponíveis. Este tipo de registrador foi implementado através de uma rotina que simula o comportamento de memórias *multi-porta*, descrito anteriormente no capítulo II.

Esta rotina deve ser divisível, ou seja, deve poder ser interrompida, de modo a permitir que os processos requisitem operações simultâneas de leitura e escrita no mesmo registrador. No caso em que dois processos tentam acessar simultaneamente um registrador atômico, um deles é forçado a esperar que o outro conclua a sua operação. Em um registrador atômico, as operações de leitura e escrita são serializadas, ou seja, são executadas segundo uma ordem determinada, uma de cada vez.

Para impedir o acesso simultâneo à memória é necessário um *bit* a mais em cada registrador (denominado *bit* de contenção) com a finalidade de indicar que o registrador já está sendo acessado.

A rotina test_and_set foi criada para efetuar este controle. Ela verifica se a célula está sendo acessada (lida ou escrita) e obtém seu

controle quando ela estiver disponível. Este procedimento simula a lógica de contenção que está presente nos *chips* de memória multi-porta, fazendo com que haja espera por parte dos processos em caso de tentativa de acesso simultâneo. As rotinas de leitura e escrita de registradores atômicos obedecem ao seguinte procedimento:

- Obtêm o controle da célula chamando a rotina test_and_set (*bit* de contenção)
- Simulam o tempo de chaveamento do barramento caso esta operação tenha sido retardada até a conclusão da operação corrente
- Executam a operação de leitura ou escrita requisitada
- Simulam o tempo de acesso da memória atômica
- Liberam o acesso à célula (tornando o *bit* de contenção = 0)

CAPÍTULO V

ANÁLISE DOS RESULTADOS

V.1 OS RESULTADOS

Os resultados obtidos podem ser encontrados na forma de gráficos ao final deste capítulo. Estes gráficos apresentam a variação do desempenho dos algoritmos implementados para solucionar o problema de Exclusão-ℓ em relação à variação do fator ℓ/n (número de recursos/número de processos).

É importante observar que o desempenho das soluções não se mantém constante com a razão ℓ/n, ou seja, o desempenho da solução com $n = 10$ e $\ell = 2$ não é igual ao obtido com $n = 20$ e $\ell = 4$ (embora $\ell/n = 1/5$ nos dois casos). Quanto maior o número de processos (n), maior é o grafo G(i) correspondente e conseqüentemente maior será o tempo de processamento gasto na seção de entrada (proporcional a n^2) para sincronizar o acesso dos processos às suas seções críticas. Por esta razão, o desempenho é reduzido com o aumento do número de processos (mantendo a relação ℓ/n fixa).

É esperado que o desempenho do sistema cresça à medida que o fator se aproxime de 1. O desempenho máximo ocorre quando o fator $\ell/n \geq 1$, pois neste caso existem recursos suficientes para todos os processos, eliminando assim a necessidade de espera na seção de entrada. Este tipo de comportamento foi confirmado pelo resultados experimentais obtidos e pode ser facilmente observado nos gráficos em que n é mantido fixo.

Os resultados obtidos com os registradores regulares foram omitidos para facilitar a visualização, uma vez que se aproximam

bastante dos obtidos com registradores *safe*. Isto ocorre porque as primitivas de manipulação das estruturas de dados de comunicação usadas (*take* e *offer*) não escrevem um valor se este for igual ao armazenado no registrador. Nesta situação particular, os registradores *safe* se comportam como regulares.

Como foi descrito no capítulo II, em caso de acesso simultâneo, os registradores *safe* e regulares são mais eficientes do que os registradores atômicos, pois não impõem espera aos processos que executam operações concorrentes. Portanto, em primeira análise, poderíamos esperar que os algoritmos baseados em tais registradores apresentassem um desempenho maior. Entretanto, os algoritmos das soluções baseadas nestes tipos de registradores mais fracos são mais complexos e conseqüentemente demandam um tempo maior de processamento para a sincronização dos processos.

Nos algoritmos baseados em registradores *safe* e regulares as primitivas de manipulação das estruturas abstratas de dados, *take* e *offer* necessitam em média de 6 operações de leitura (e escrita) sobre os *bits* da estrutura, enquanto que para os registradores atômicos são necessárias apenas 4 operações.

De modo a facilitar a comparação dos desempenhos das soluções, os tempos de acesso das operações não-concorrentes dos registradores **SWMR** foram feitos iguais através do ajuste das rotinas de simulação. A duração das operações de leitura e escrita concorrentes em registradores atômicos é variável, pois elas devem ser realizadas em série, ou seja, uma de cada vez. Os processos que tentam acessar este registrador simultaneamente são forçados a esperar a conclusão das outras operações correntes.

Em um sistema com processos que se comunicam através de

registradores atômicos compartilhados, o atraso total gerado pela espera imposta aos processos é proporcional ao percentual de operações simultâneas de leitura e escrita no mesmo registrador.

Assim, seria razoável esperarmos que os algoritmos que utilizam registradores *safe* e regulares sejam mais eficientes nas situações em que o número de operações de leitura e escrita concorrentes nos registradores compartilhados seja grande, e que os algoritmos baseados em registradores atômicos apresentem um desempenho melhor nos casos em que o percentual de operações concorrentes seja baixo.

Podemos observar atráves dos gráficos ao final deste capítulo, que, de um modo geral, a solução do problema de Exclusão-ℓ baseada em registradores atômicos, implementada nesta tese, apresenta um desempenho maior que as outras duas soluções. Isto ocorre porque o tempo de processamento gasto para a manipulação das estruturas de dados utilizadas nas soluções *safe* e regular é maior que o tempo gasto pela espera imposta aos processos em caso de acesso simultâneo aos registradores atômicos.

A justificativa para este fato é que, na maioria das situações ao qual o algoritmo foi submetido, o número de operações de leitura e escrita concorrentes varia de 2% a 6% do total de operações realizadas.

Como neste caso a quantidade de operações simultâneas é baixa, a ineficiência introduzida com o uso de registradores atômicos não é muito significativa, fazendo com que esta solução apresente um desempenho melhor que o das demais.

Os gráficos em que ℓ é fixo demonstram o raciocínio descrito anteriormente. No caso em que $T_r = T_c = P$, o número de operações concorrentes varia de 2% a 35%, fazendo com que as curvas de desempenho dos algoritmos se cruzem em $\ell/n \approx 65\%$ (número de operações concorrentes

: 12%). Para ℓ/n < 65% o algoritmo baseado em registradores atômicos é ais eficiente. O mesmo ocorre nos gráficos em que $T_r = T_c = G$ e $T_r = P$ e $_c = G$. Nestes dois gráficos, os pontos em que as curvas de desempenho e cruzam são, respectivamente, ℓ/n = 60% e ℓ/n = 80%.

Ao contrário dos gráfico em que n é fixo, os gráficos com ℓ fixo presentam curvas bastante diferentes para as 4 combinações possíveis e T_r e T_c. Nos gráficos em que $T_r = G$, o desempenho cresce rapidamente om a diminuição do número de processos no sistema (aumento da razão $/n$, pois ℓ é fixo), quase atingindo seu valor máximo quando $\ell/n \approx 60\%$. sto ocorre porque os processos passam grande parte de seu tempo de xecução nas suas seções Restantes, não havendo assim muita competição a seção de entrada para executarem suas seções críticas. Esta disputa ara entrarem em suas seções críticas só começa a ser notada quando o úmero de processos ultrapassa o dobro do número máximo de processos ue podem executar suas seções críticas simultaneamente (número de ecursos).

É interessante observar que no gráfico em que $T_r = P$ e $T_c = G$, a elação entre o desempenho e o fator ℓ/n é linear. Este é o caso xtremo em que os processos estão constantemente competindo para ntrarem em suas seções críticas. Quanto mais processos fizerem parte o sistema, menor será o desempenho total, pois os processos gastam a aior parte do seu tempo de processamento executando o algoritmo de incronização na seção de entrada.

.2 GRÁFICOS DE DESEMPENHO DAS SOLUÇÕES

Para facilitar a interpretação dos dados numéricos obtidos na xecução das três soluções do problema de Exclusão-ℓ, apresentamos stes dados na forma de gráficos de desempenho em função da razão ℓ/n

(número de recursos/número de processos).

Estes gráficos são apresentados, a seguir, na seguinte ordem:

- Gráficos com ℓ fixo ($\ell = 2$) para registradores atômicos, regulares e *safe*:
  - T_r = P e T_c = P
  - T_r = P e T_c = P (Logarítmico)
  - T_r = P e T_c = G
  - T_r = G e T_c = P
  - T_r = G e T_c = G

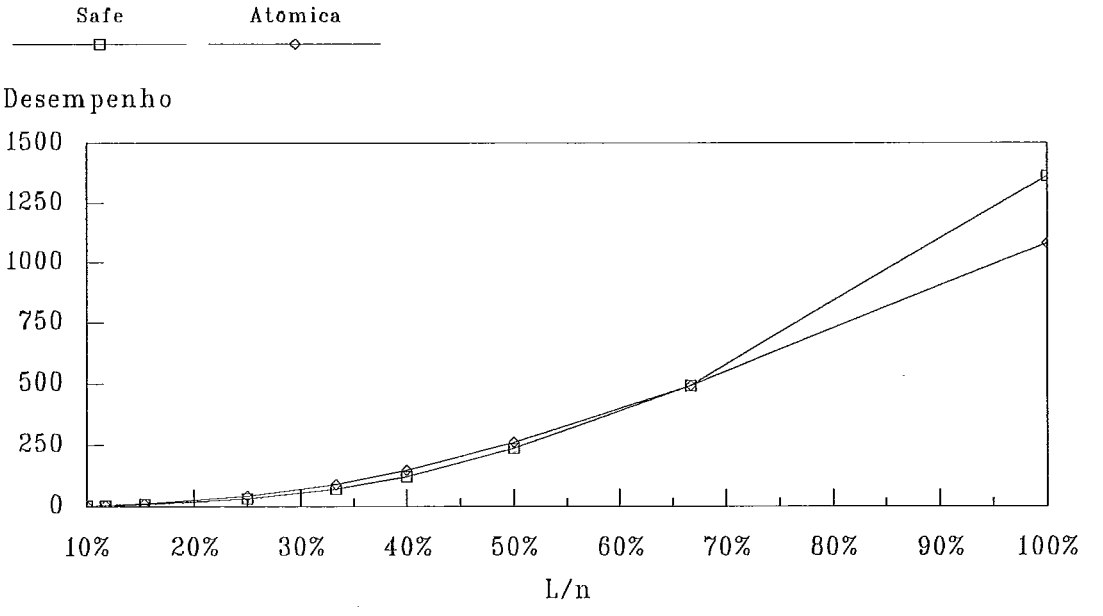
- Gráficos com n fixo ($n = 20$) para registradores atômicos, regulares e *safe*:
  - T_r = P e T_c = P
  - T_r = P e T_c = G
  - T_r = G e T_c = P
  - T_r = G e T_c = G

- Gráficos de registradores *safe* para as 4 combinações dos tempos T_r e T_c (P x P, P x G, G x P, G x G):
  - Com ℓ fixo ($\ell = 2$)
  - Com n fixo ($n = 20$)

- Gráficos de registradores atômicos para as 4 combinações dos tempos T_r e T_c (P x P, P x G, G x P, G x G):
  - Com ℓ fixo ($\ell = 2$)
  - Com n fixo ($n = 20$)

Exclusão-1

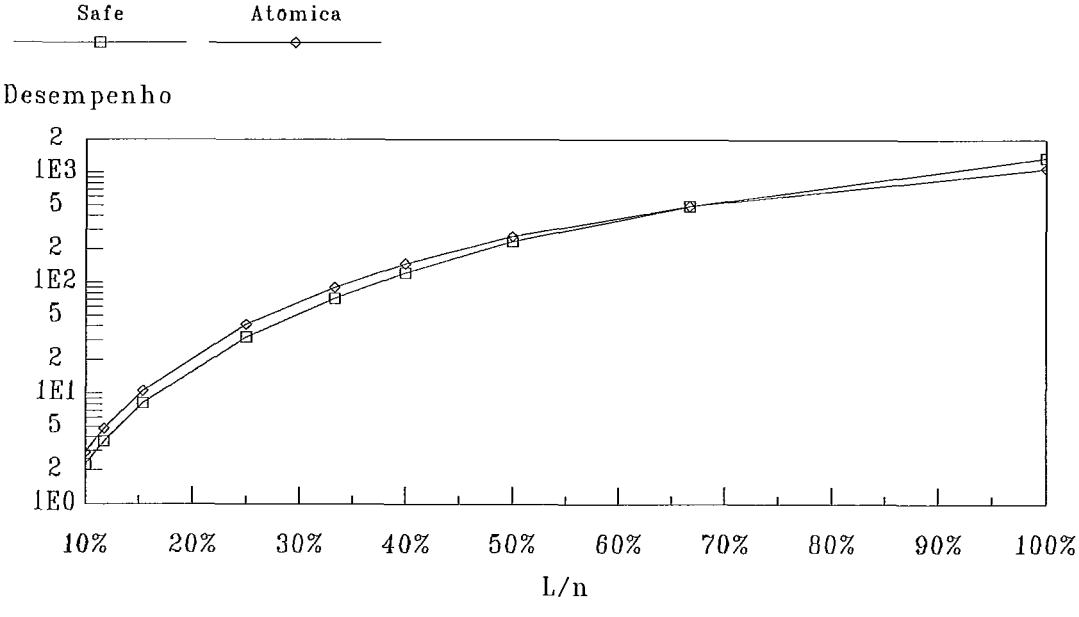
L fixo (2)



Processos: 1..20 Tr = P Tc = P

Exclusão-1

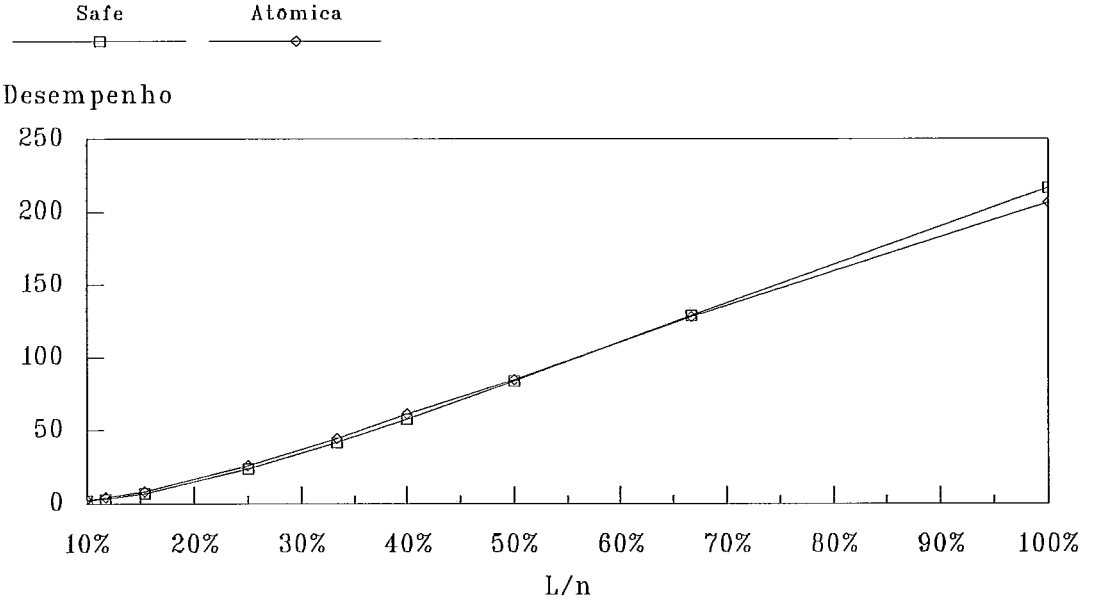
L fixo (2)



Processos: 1..20 Tr = P Tc = P

Exclusão-1

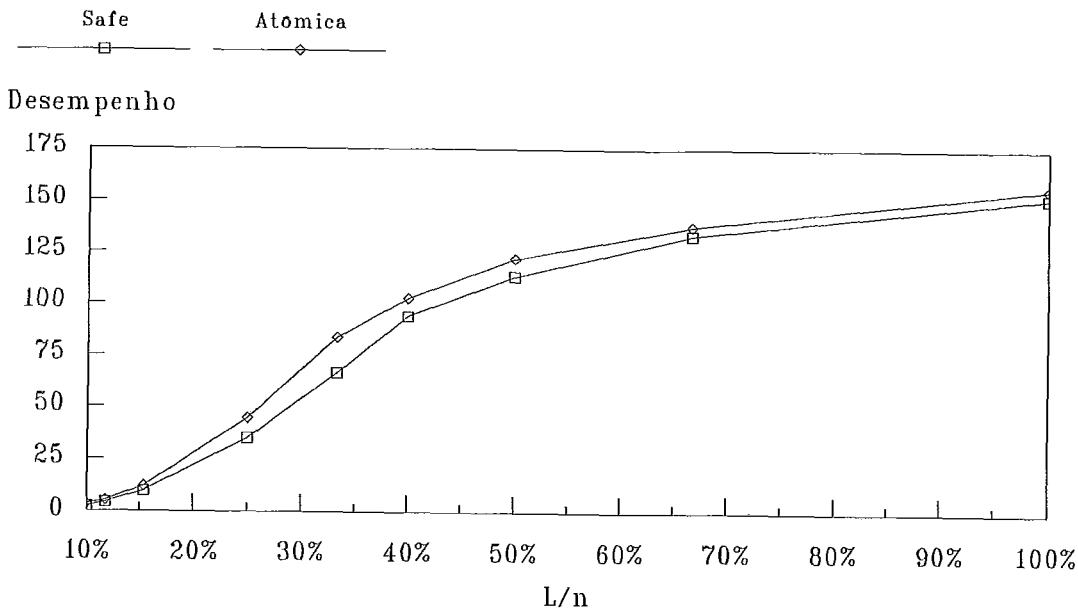
L fixo (2)



Processos: 1..20 Tr = P Tc = G

Exclusão-1

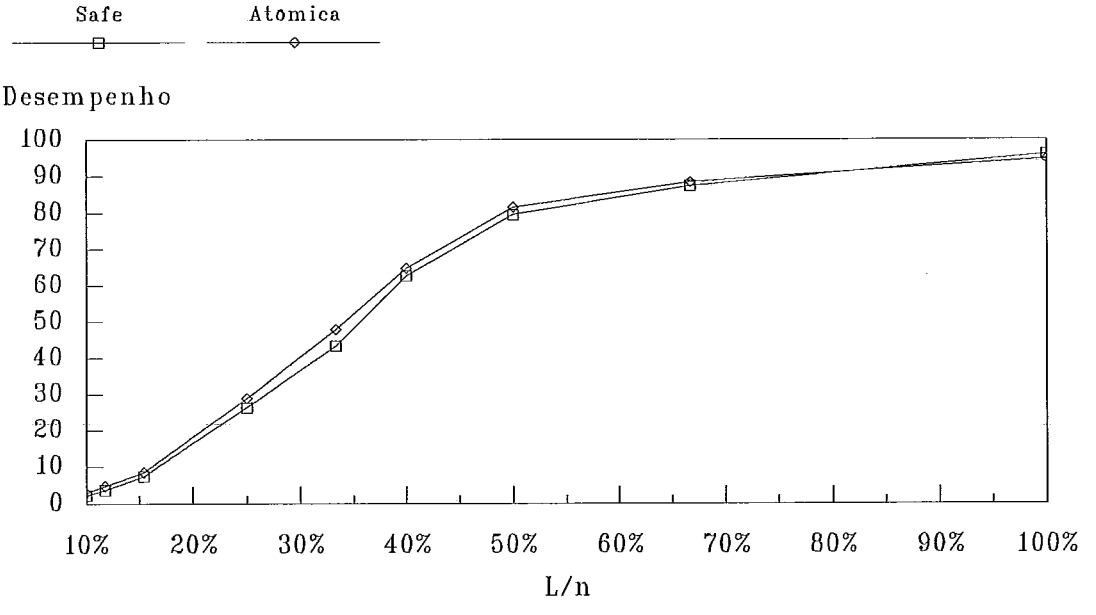
L fixo (2)



Processos: 1..20 Tr = G Tc = P

Exclusão-1

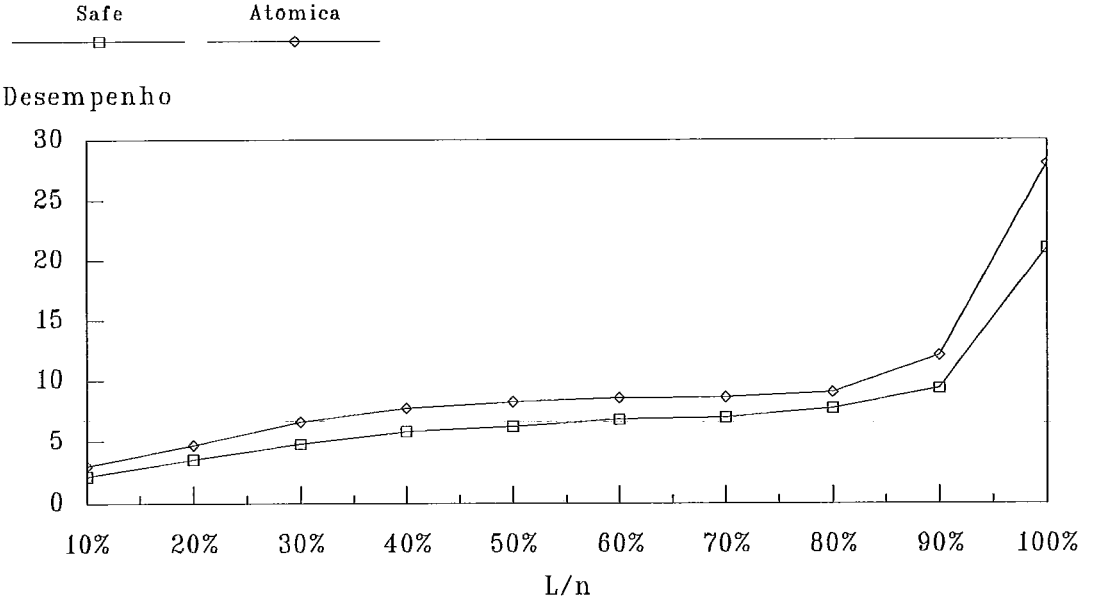
L fixo (2)



Processos: 1..20 Tr = G Tc = G

Exclusão-1

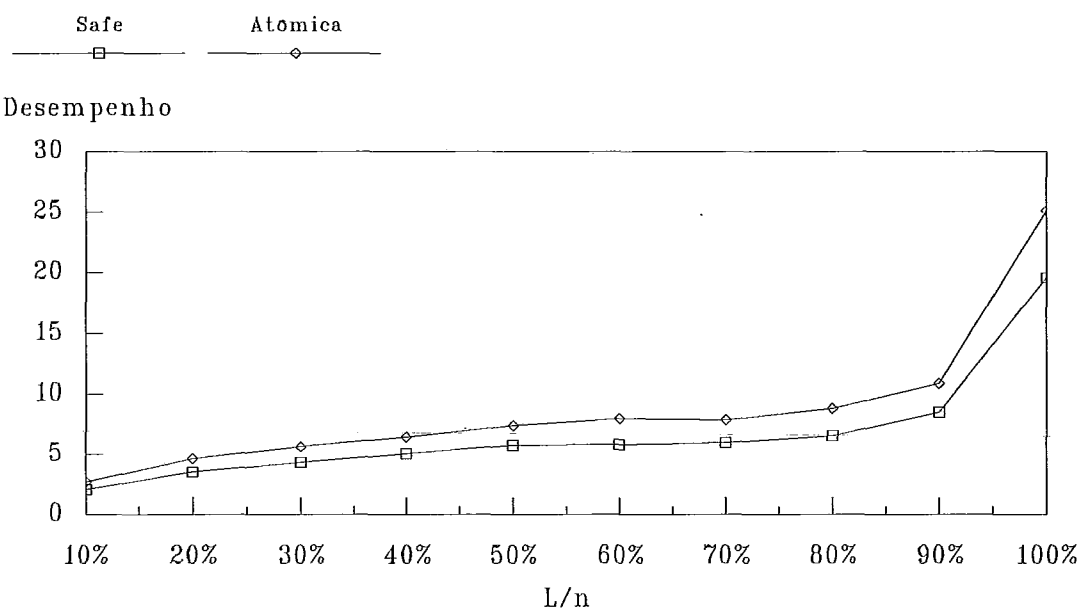
N fixo (20)



Processos: 20 Tr = P Tc = P

Exclusão-1

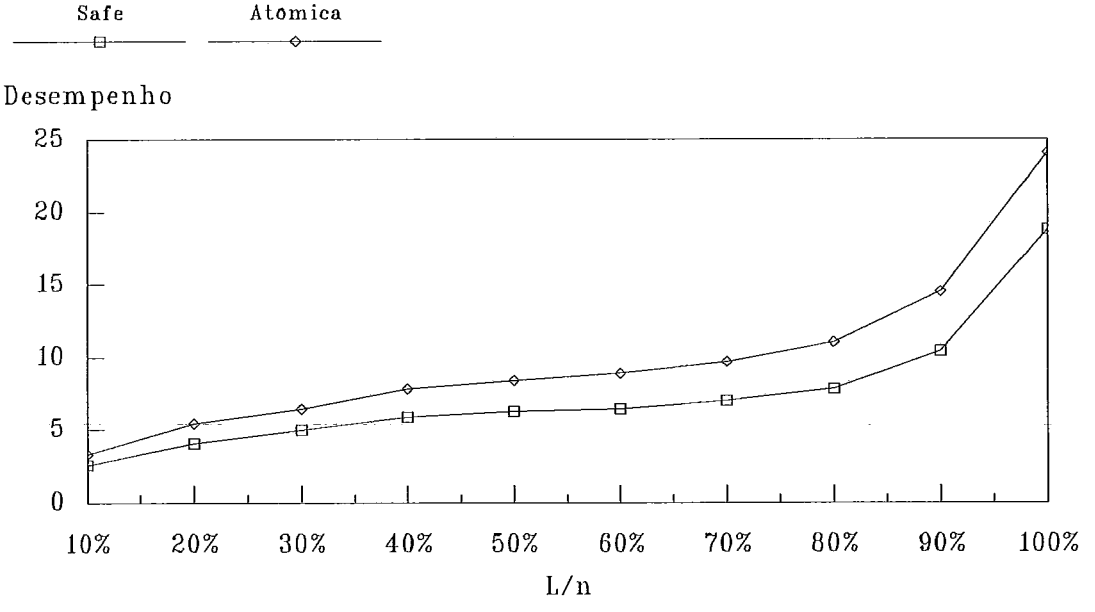
N fixo (20)



Processos: 20 Tr = P Tc = G

Exclusão-1

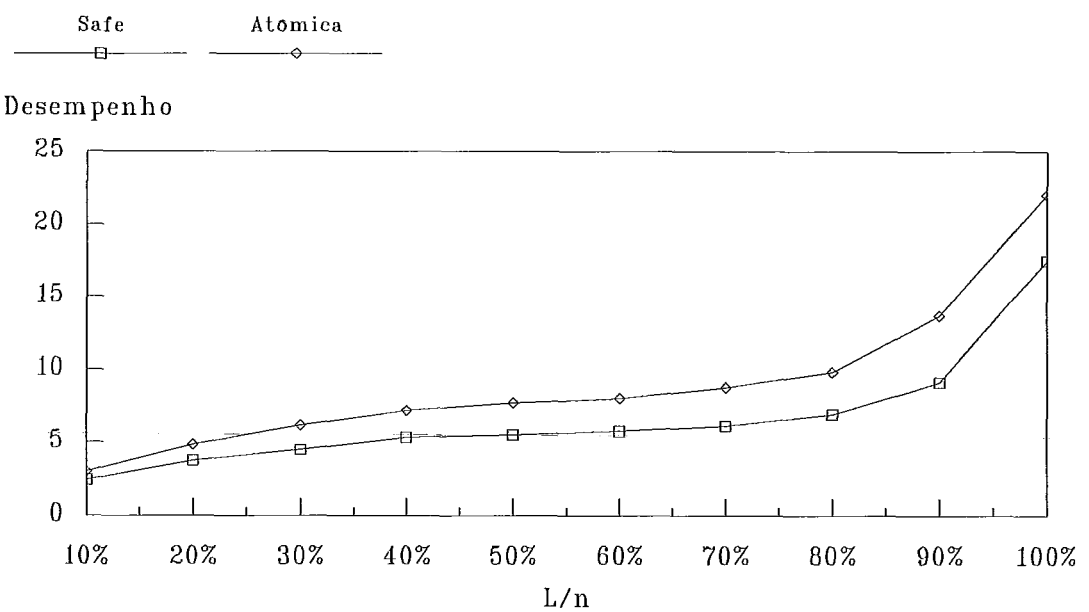
N fixo (20)



Processos: 20 Tr = G Tc = P

Exclusão-1

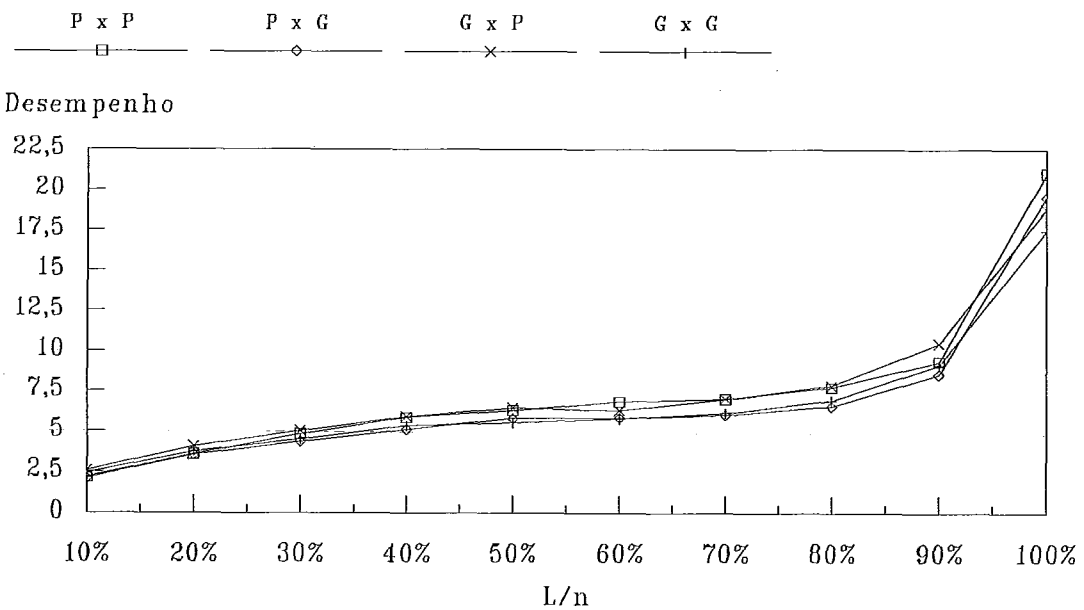
N fixo (20)



Processos: 20 Tr = G Tc = G

Exclusão-1

N fixo (20)

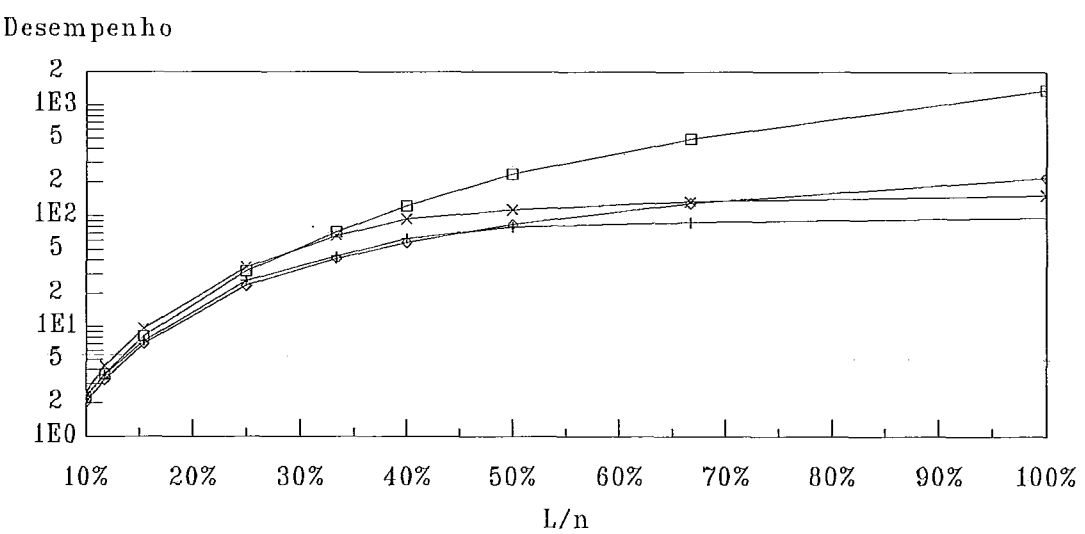


Memória = Safe

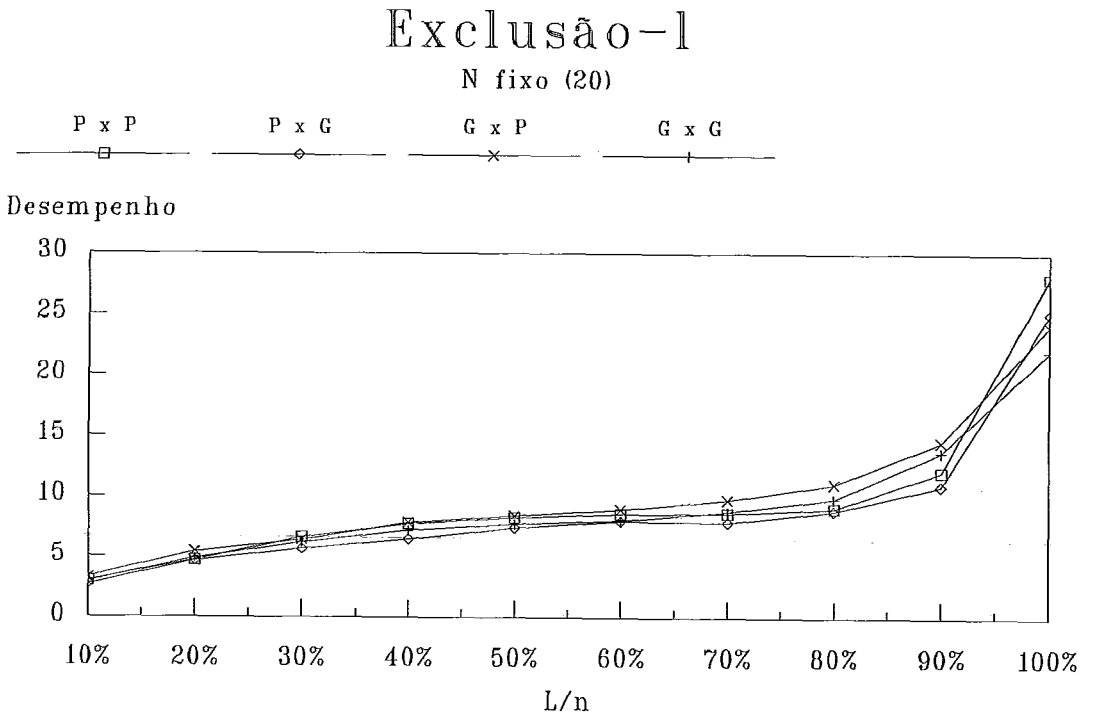
Exclusão-1

L fixo (2)

P x P P x G G x P G x G



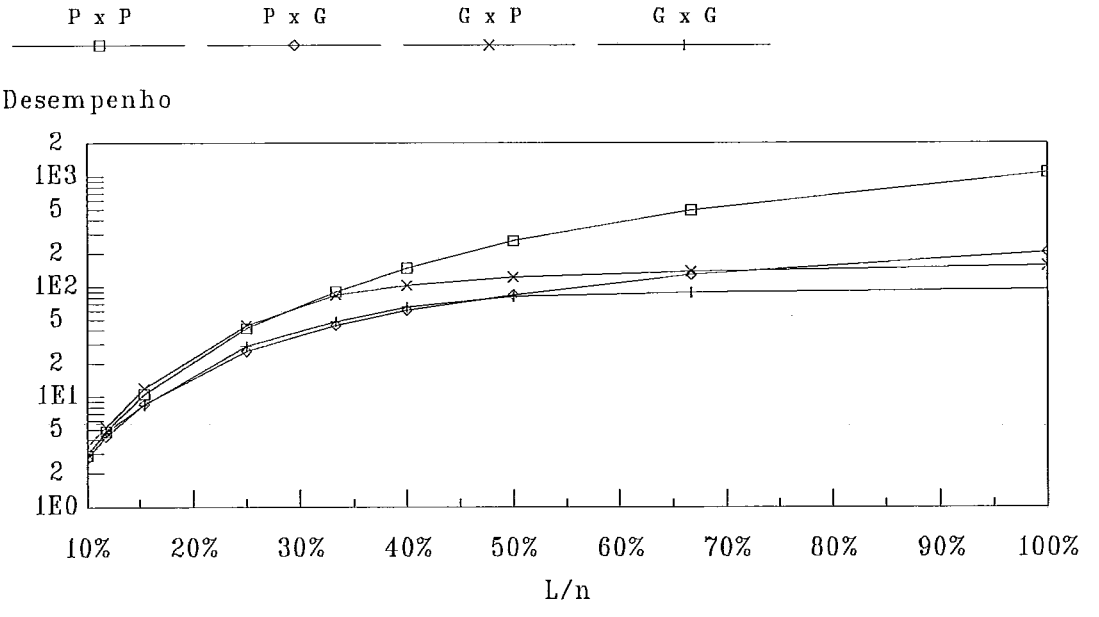
Processos: 1..20 Memória = Safe



Memória = Atomica

Exclusão-1

L fixo (2)



Processos: 1..20 Memória = Atomica

APÊNDICE

LISTAGEM DOS PROGRAMAS

Este apêndice contém a listagem dos programas que foram desenvolvidos para implementar as soluções do problema de Exclusão-ℓ, descritas no capítulo III. Todos os módulos foram escritos em linguagem C, com exceção do *kernel* do sistema que, por questões de eficiência, teve de ser codificado em *Assembly 8086*. O programa é composto dos seguintes módulos:

- KERNEL.ASM - Implementa um núcleo preemptivo para permitir a execução de vários processos concorrentes em um único processador.
- TASK.C - Implementa a interface para a seleção de parâmetros, assim como a inicialização do sistema e o armazenamento dos resultados.
- PROCS.H - Contém as definições das constantes utilizadas pelos processos do sistema.
- TASK.H - Contém o algoritmo para a solução do problema de Exclusão-ℓ, as primitivas de manipulação das estruturas de dados de comunicação e as rotinas que simulam os registradores **SWMR**. É composto de uma parte dependente e outra independente do tipo de registrador utilizado.
- PROC0.C .. PROCn.C - Contém a rotina principal dos processos.

```
/*************************************************************************
*
*  PROCS.H - Definição das contantes utilizadas pelos processos
*            PROC0 .. PROCn que executam o algoritmo de Exclusão-ℓ
*
*************************************************************************/


/* Definição do tipo de memória utilizada */

#define SAFE            0
#define REGULAR         0
#define ATOMICO         1

#define MAX_PROCS       50

/* Estados possíveis para uma estrutura do tipo FORK */

#define OFER_I          0
#define EM_USO_J        1
#define OFER_J          2
#define EM_USO_I        3
#define I__J            0
#define J__I            1

#define TRUE            0xFF
#define FALSE           0

#define FP_OFF(p)       *((int *)&p)
#define FP_SEG(p)       *((int *)&p + 1)

#if (SAFE || REGULAR)
/*  Definição da estrutura de dados do tipo FORK */

typedef struct {
                  char fi;
                  char fj;
                  char wqi;
                  char wpi;
                  char wqj;
                  char wpj;
                } FORK;
#else
typedef char FORK;
#endif
```

```
/*************************************************************************
*
*  TASK.C - Módulo de interface e inicialização do sistema
*
*
*************************************************************************/


#include "stdio.h"
#include "string.h"
#include "procs.h"
#include "stdlib.h"

#define STACK_SIZE      5000            /* 512 */
#define FLAGS           0x0200          /* Interrupt = EI  */
#define PERIODO         0.838096E-3     /* Período minimo do 8253
                                           no IBM-PC) */
#define NVEZES          10              /* Número de medidas obtidas
                                           para cada teste */

/* Estruturas utilizadas no algoritmo de Exclusão-ℓ */

/* Forks  para  implementar a Exclusão-ℓ */
FORK  G_ARROW[MAX_PROCS][MAX_PROCS];

/* Forks para a ausência de lockout */
FORK ID[MAX_PROCS][MAX_PROCS];

char x[MAX_PROCS];

/* Definição do Bloco de Controle das tarefas */

struct TCBN {
                struct TCBN *Link;
                int    iax;
                int    ibx;
                int    icx;
                int    idx;
                int    isi;
                int    idi;
                int    ibp;
                int    isp;
                int    iss;
                int    ies;
                int    ids;
                int    iflags;
                int    iip;
                int    ics;
            };

unsigned _STACK = 15000;        /* Tamanho da Stack do programa */

unsigned int total_crit_section[MAX_PROCS] = { 0, 0 , 0 , 0 , 0 };

unsigned int counter;     /* Divisor da freqüência de troca de tarefas*/

unsigned int delay;       /* Simula o tempo de acesso de uma
```

```
unsigned long ticks = 0L;       /* Contador de ticks */
unsigned long ticks2 = 0L;      /* Contador de ticks */
unsigned long total_ticks;      /* Duração do teste em ticks */

double time_slice, media, time_total, temp;

struct TCBN tcb[MAX_PROCS+1];

int task0(), task1(), task2(), task3(), task4(),
    task5(), task6(), task7(), task8(), task9(),
    task10(), task11(), task12(), task13(), task14(),
    task15(), task16(), task17(), task18(), task19(),
    task20(), task21(), task22(), task23(), task24(),
    task25(), task26(), task27(), task28(), task29(),
    task30(), task31(), task32(), task33(), task34(),
    task35(), task36(), task37(), task38(), task39(),
    task40(), task41(), task42(), task43(), task44(),
    task45(), task46(), task47(), task48(), task49();

struct TCBN *sys_tsk = &tcb[0];

int (*procs[])() = {
                        &task0, &task1, &task2, &task3, &task4 ,
                        &task5, &task6, &task7, &task8, &task9,
                        &task10, &task11, &task12, &task13, &task14,
                        &task15, &task16, &task17, &task18, &task19,
                        &task20, &task21, &task22, &task23, &task24,
                        &task25, &task26, &task27, &task28, &task29,
                        &task30, &task31, &task32, &task33, &task34,
                        &task35, &task36, &task37, &task38, &task39,
                        &task40, &task41, &task42, &task43, &task44,
                        &task45, &task46, &task47, &task48, &task49
                    };

char *stack;                /* ponteiro da stack */
unsigned int ss;

/* Número de processos máximo na seção crítica simultaneamente */

char l_Exclusion;
char NPROCS;                /* Número de processos no sistema */
int rep;                    /* Contador de repetições das medidas */
double media_total;         /* Média dos resultados das execuções */

int video_ptr = 0;          /* Utilizado na depuração */

int crit_section = 0;       /* Contador do número de processos na seção
                               crítica */

int tempo = 1;              /* Tempo de teste default = 1s */

/* Valor dos loops das seções restantes dos processos */
unsigned int tr[MAX_PROCS];

/* Valor dos loops das seções críticas dos processos */
unsigned int tc[MAX_PROCS];

int tr0 = 1;
```

```
int mask_tr = 1;

/* Contadores de leitura e escrita (total e concorrente) */
unsigned long read_total, write_total, write_concorrente,
              read_concorrente;

main(argc,argv)
char *argv[], argc;
{
 int i, j;

 printf("%c", '"');

#if SAFE
 printf(" SAFE ");
#endif

#if REGULAR
 printf(" REGULAR ");
#endif

#if ATOMICO
 printf(" ATOMICA ");
#endif


/* Obtêm os parâmetros do sistema na linha de comando */

 if (argc != 1) {
   counter = atoi(argv[1]);
 }

 if (argc > 2) {
   delay = atoi(argv[2]);
 }

 if (argc > 3) {
   tempo = atoi(argv[3]);
 }

 if (argc > 4) {
   tr0 = atoi(argv[4]);
 }
 if (argc > 5) {
   tc0 = atoi(argv[5]);
 }

/* Obtém o número de processos no sistema */

 if (argc > 6) {
   NPROCS = atoi(argv[6]);
 }

 temp = (double)tr0 / 10;
 for(mask_tr = 1; mask_tr < temp; mask_tr *= 2);

 temp = (double)tc0 / 10;
 for(mask_tc = 1; mask_tc < temp; mask_tc *= 2);
```

```
 if (argc > 7) {
   l_Exclusion = atoi(argv[7]);
 }

  time_slice = (double)counter * PERIODO;
  total_ticks = (long)( (double)tempo*1000/time_slice)*NPROCS - 1;

  printf("N= %d L= %d Tr=%d Tc=%d Delay=%d Slice= %.2g us Tempo=%u s",
          NPROCS, l_Exclusion, tr0, tc0,delay, time_slice*1000,tempo);
  printf(" Mr= %d Mc= %d", mask_tr , mask_tc );
  printf("%c", '"');

/* Retorna de número de recursos for 0 */
  if (l_Exclusion == 0){
    printf(" \n \n");
    exit(0);
  }

/* Aloca espaço para as stacks dos processos */

  if ( (stack = malloc(NPROCS * STACK_SIZE)) == NULL ) {
   printf("Erro na alocação da stack dos processos\n");
   exit(3);
  }

  for(rep=0; rep < NVEZES; rep++)
   {
    ticks = ticks2 = 0;
    media = crit_section = 0;
    for(i=0; i < NPROCS; i++)
     for(j=0; j < NPROCS; j++) G_ARROW[i][j] = 0;

    ss = FP_SEG(stack);

/* Inicializa tcbs */

    for( i=0; i < NPROCS; i++){

     if (i < NPROCS -1) tcb[i].Link = &tcb[i+1];   /* Liga as TCBs em
                                                      uma cadeia */

/* Aponta TCB para o início das tarefas */

     *((char **)(&tcb[i].iip)) = (char *)procs[i];

/* Inicializa a stacks dos processos nas TCBs */

     ss += STACK_SIZE/16;
     tcb[i].isp = STACK_SIZE - 2;
     tcb[i].iss = ss;

/* Inicializa o segmento de dados inicial dos processos */

     tcb[i].ids = dseg();
     tcb[i].iflags = FLAGS;
    }
```

```
 /* Instala o kernel na interrupção de relógio */

    install();

 /* Executa a primeira tarefa */

    task0();

 /* Imprime os resultados */

    for(i = 0; i < NPROCS; i++) {
     media += total_crit_section[i];
    }
    time_total = ticks2*(time_slice/1000);

/*   printf(", %g , %g,", (double)(l_Exclusion)/NPROCS,
         media/time_total );  */

    media_total += media/time_total;
   }
   printf(", %g , %g,", (double)(l_Exclusion)/NPROCS,
          media_total/NVEZES );


#if ATOMICO
   printf(" %.4g, %.4g,\n",
         (double)read_concorrente/(double)read_total*100,
         (double)write_concorrente/(double)write_total*100);
#else
   printf("\n");
#endif

}


Imprime(str)    /* Imprime caracteres diretamente na tela do micro */
char *str;      /* Utilizado para depuração */
{
 char *p;

  while( *str != NULL )
  {
   FP_OFF(p) = video_ptr;
   FP_SEG(p) = 0xOB800;

   *p = *str++;
   p = p + 2;
   video_ptr = FP_OFF(p);
   if (video_ptr >= 80*24*2 ) video_ptr = 0;
  }
}


/*********************************************************************
*
*  Tarefas do sistema
*
*********************************************************************/
```

```
{
   proc0();                          /* Chama o processo 0 */
}

task1()
{
   proc1();                          /* Chama o processo 1 */
}

task2()
{
   proc2();                          /* Chama o processo 2 */
}

task3()
{
   proc3();                          /* Chama o processo 3 */
}

task4()
{
   proc4();                          /* Chama o processo 4 */
}

task5()
{
   proc5();                          /* Chama o processo 5 */
}

task6()
{
   proc6();                          /* Chama o processo 6 */
}

task7()
{
   proc7();                          /* Chama o processo 7 */
}

task8()
{
   proc8();                          /* Chama o processo 8 */
}

task9()
{
   proc9();                          /* Chama o processo 9 */
}

task10()
{
   proc10();                         /* Chama o processo 10 */
}

task11()
{
   proc11();                         /* Chama o processo 11 */
}
```

```
{
   proc12();                    /* Chama o processo 12 */
}

task13()
{
   proc13();                    /* Chama o processo 13 */
}

task14()
{
   proc14();                    /* Chama o processo 14 */
}

task15()
{
   proc15();                    /* Chama o processo 15 */
}

task16()
{
   proc16();                    /* Chama o processo 16 */
}

task17()
{
   proc17();                    /* Chama o processo 17 */
}

task18()
{
   proc18();                    /* Chama o processo 18 */
}

task19()
{
   proc19();                    /* Chama o processo 19 */
}

task20()
{
   proc20();                    /* Chama o processo 20 */
}

task21()
{
   proc21();                    /* Chama o processo 21 */
}

task22()
{
   proc22();                    /* Chama o processo 22 */
}

task23()
{
   proc23();                    /* Chama o processo 23 */
}
```

```
{
   proc24();                    /* Chama o processo 24 */
}

task25()
{
   proc25();                    /* Chama o processo 25 */
}

task26()
{
   proc26();                    /* Chama o processo 26 */
}

task27()
{
   proc27();                    /* Chama o processo 27 */
}

task28()
{
   proc28();                    /* Chama o processo 28 */
}

task29()
{
   proc29();                    /* Chama o processo 29 */
}

task30()
{
   proc30();                    /* Chama o processo 30 */
}

task31()
{
   proc31();                    /* Chama o processo 31 */
}

task32()
{
   proc32();                    /* Chama o processo 32 */
}

task33()
{
   proc33();                    /* Chama o processo 33 */
}

task34()
{
   proc34();                    /* Chama o processo 34 */
}

task35()
{
   proc35();                    /* Chama o processo 35 */
}
```

```
{
   proc36();                    /* Chama o processo 36 */
}

task37()
{
   proc37();                    /* Chama o processo 37 */
}

task38()
{
   proc38();                    /* Chama o processo 38 */
}

task39()
{
   proc39();                    /* Chama o processo 39 */
}

task40()
{
   proc40();                    /* Chama o processo 40 */
}

task41()
{
   proc41();                    /* Chama o processo 41 */
}

task42()
{
   proc42();                    /* Chama o processo 42 */
}

task43()
{
   proc43();                    /* Chama o processo 43 */
}

task44()
{
   proc44();                    /* Chama o processo 44 */
}

task45()
{
   proc45();                    /* Chama o processo 45 */
}

task46()
{
   proc46();                    /* Chama o processo 46 */
}

task47()
{
   proc47();                    /* Chama o processo 47 */
}
```

```
{
   proc48();                    /* Chama o processo 48 */
}

task49()
{
   proc49();                    /* Chama o processo 49 */
}
```

```
/*************************************************************************
*
* TASK.H  - Rotinas comuns aos processos PROC0..PROCn para simular o
*           acesso às memórias do tipo SAFE, REGULAR e ATOMICA
*
*************************************************************************/


extern FORK G_ARROW[MAX_PROCS][MAX_PROCS];
extern FORK ID[MAX_PROCS][MAX_PROCS];
extern char NPROCS;
extern char l_Exclusion;

extern char x[MAX_PROCS];
extern unsigned int total_crit_section[MAX_PROCS];

#define GRAPH   G_ARROW

extern int video_ptr;
extern int crit_section;
extern int tr0;
extern int tc0;
extern long ticks2;
extern long total_ticks;
extern int mask_tr;
extern int mask_tc;

static id_str[] = {'0' + PROC_NUM, 0};

static char visitado[MAX_PROCS];
static char PROC_ID[MAX_PROCS];

process()
{
 int j;
 extern int random;
 int tr_time, tc_time;

   for(;;) {

/* Verifica se esgotou o tempo de execução */

   if ((ticks2 >= total_ticks) && (PROC_NUM == 0)){
      retira();
        return;
    }

 /* Execução da seção restante */

     tr_time = tr0 + (random & mask_tr);

     for( j = 0; j < tr_time; j++);

     x[PROC_NUM] = TRUE;

 /* Redireciona as estruturas para os processos adjascentes */
```

```
L:
     increment_your_id();
     observe_id();
     oracle();
     if ( R() > l_Exclusion ) goto L;

 /*   Seção Crítica   */

        total_crit_section[PROC_NUM]++;
        crit_section++;

        prt_crit();

#if IMP
        Imprime(id_str);
#endif

/*  Execução da Seção Crítica */

     tc_time = tc0 + (random & mask_tc);

     for( j = 0; j < tc_time; j++);

/*   Testa o número de processos na seção crítica */

     prt_crit();
     crit_section--;

/*   Sai de seção crítica  */

     initialize_your_id();
     x[PROC_NUM] = FALSE;

   }
}


/*************************************************************************
*
* Área das rotinas independentes do tipo de registrador utilizado
*
*************************************************************************/


/*
*  R() - retorna a cardinalidade do conjunto de alcançabilidade lido
*         pelo processo PROC_NUM
*/

static R()
{
 int i,j, cardinalidade;
 char G[MAX_PROCS][MAX_PROCS];

/* Reseta a variável de visita */

  for( i =0; i < NPROCS; i++) visitado[i] = FALSE;
```

```
  for(i = 0; i < NPROCS- 1; i++)
    for(j = i + 1; j < NPROCS; j++)
       G[i][j] = read_arrow(&G_ARROW[i][j]);

 /* Obtêm a cardinalidade do subgrafo G'(i) */

  cardinalidade = conta(PROC_NUM, G);

  return(cardinalidade);
}


static conta(i, G)
char i;
char *G;
{
 int total, j;

 visitado[i] = TRUE;
 total = 0;

  for(j = 0; j < NPROCS; j++) {

   if ( (j==i) || !x[j] || visitado[j] ) continue;
   if (i < j){
        if (*(G + i*MAX_PROCS + j) != I__J ) continue;
   }
   else if (*(G + j*MAX_PROCS + i) != J__I ) continue;

   total = total + conta(j, G);
  }

  return(total+1);
}


static read_arrow(arrow_ij)
char *arrow_ij;
{
   switch( read_fork(arrow_ij) )
    {
      case OFER_J:
        case EM_USO_I:
          return(J__I);

        default:
          return(I__J);
    }
}

static oracle()
{
 char j, id;

   id = PROC_ID[PROC_NUM];

 /* Redireciona as setas para os processos que possuem ids maiores */
```

```
      if (j == PROC_NUM) continue;
      if (PROC_ID[j] > id || !x[j]) redirect(PROC_NUM,j,GRAPH);
        else if (PROC_ID[j] == id && j > PROC_NUM)
                                    redirect(PROC_NUM,j,GRAPH);
   }
}

static Imprime(str)
char *str;
{
 char *p;

  while( *str != NULL )
  {
   FP_OFF(p) = video_ptr;
   FP_SEG(p) = 0xOB800;

   *p = *str++;
   p = p + 2;
   video_ptr = FP_OFF(p);
   if (video_ptr >= 80*24*2 ) video_ptr = 0;
  }
}

static increment_your_id()
{
 char *id_ij;
 int i,j;

  i = PROC_NUM;

  /* Tenta obter todos os forks que estão disponíveis */

  for(j = 0; j < NPROCS; j++) {

    if (i == j) continue;
    if (i < j) {
       id_ij = (char *)ID + (i*MAX_PROCS + j) * sizeof(FORK);
       take_i(id_ij);
    } else {
       id_ij = (char *)ID + (j*MAX_PROCS + i) * sizeof(FORK);
       take_j(id_ij);
    }
  }
}

static initialize_your_id()
{
 char *id_ij;
 int i,j;

  i = PROC_NUM;

  /* Libera todos os FORKS que possui */

  for(j = 0; j < NPROCS; j++) {

    if (i == j) continue;
```

```
        offer_i(id_ij);
      } else {
        id_ij = (char *)ID + (j*MAX_PROCS + i) * sizeof(FORK);
        offer_j(id_ij);
      }
    }
}

static observe_id()
{
 int i, j, count;

 for(i = 0; i < NPROCS; i++) {
   count = 0;

   for(j = 0; j < NPROCS; j++) {

     if (i == j) continue;
     if (i < j) {
       if (read_fork(&ID[i][j]) == EM_USO_I) count++;
     }
     else
       if (read_fork(&ID[j][i]) == EM_USO_J) count++;
   }

   PROC_ID[i] = count;
 }
}

static prt_crit()
{
  if (crit_section > 1_Exclusion) {
    retira();
    Imprime("Erro: Excesso de processo na seção crítica");
X:  goto X;
  }
}


/*************************************************************************
*
* Área das rotinas dependentes do tipo de memória (atômica, regular ou
* safe)
*
*************************************************************************/


#if REGULAR     /* Rotinas para simular memórias do tipo REGULAR */

static redirect(i, j, graph)
char *graph;
char i,j;
{
  char *arrow_ij;

   if (i==j) return;

 /* O processo i deve ser sempre menor que j */
```

```
    arrow_ij = graph + (i*MAX_PROCS  + j) * sizeof(FORK);
    take_i(arrow_ij);
    offer_i(arrow_ij);
  }
  else {
    arrow_ij = graph + (j*MAX_PROCS + i) * sizeof(FORK);
    take_j(arrow_ij);
    offer_j(arrow_ij);
  }
}


static read_fork(f)
FORK *f;
{
  if (read_k(&f->fi) == read_k(&f->fj)) {          /* if (F == 1)  */
    if (read_k(&f->wqi)) return(EM_USO_I);
                  else  return(OFER_I);
  }
  else {
    if (read_k(&f->wqj)) return(EM_USO_J);
                 else  return(OFER_J);
  }
}

static take_i(f)
FORK *f;
{
   if (read_fork(f) == OFER_J){
     write_k(&f->wpi,TRUE);                        /* wi = TRUE */
     write_k(&f->wqi,TRUE);                        /* F = 1 */
     write_k(&f->fi, f->fj)
   }
}

static take_j(f)
FORK *f;
{
   if (read_fork(f) == OFER_I){
     write_k(&f->wpj, TRUE);
     write_k(&f->wqj, TRUE);                       /* wj = TRUE */
     if (read_k(&f->fi) == read_k(&f->fj))
                   write_k(&f->fj, !f->fj);        /* F = 0 */
   }
}


static offer_i(f)
FORK *f;
{
   if (read_fork(f) == EM_USO_I){
      write_k(&f->wpi, FALSE);                     /* wi = FALSE */
      write_k(&f->wqi, FALSE);
   }
}

static offer_j(f)
```

```
   if (read_fork(f) == EM_USO_J){
      write_k(&f->wpj, FALSE);                    /* wj = FALSE */
      write_k(&f->wqj, FALSE);
   }
}

static read_k(f)
char *f;
{
 extern int random;
 char r1,r2,i;

/* As chamadas à rotina enable(), assim como as instruções duplicadas,
   têm a função de manter o tempo gasto pela rotina igual às operações
   de leitura e escrita do tipo SAFE e ATÔMICO */

 enable();
 r1 = *f & 1;                  /* Lê o bit atômico pela primeira vez */
 r1 = *f & 1;
 enable();

/* Simula o acesso à memória */
 for(i=0 ; i < 10 ; i++);

 enable();
 r2 = *f & 1;                  /* Lê o bit atômico novamente */
 r2 = *f & 1;
 enable();

/* Se leituras r1 e r2 forem diferentes a rotina escolhe uma delas */

 if (random & 1) return((int)r1);  /* Retorna o primeiro resultado */
         else    return((int)r2);  /* Retorna o segundo resultado */
}

static write_k(f, value)
char *f;
char value;
{
 char wr;

/* A função do código abaixo é de igualar o tempo de acesso das
   operações de leitura e escrita. Não tem nenhuma outra função !! */

  disable();                      /* Desabilita interrupções */
  wr = *f & 1;
  wr = *f;
  enable();                       /* Habilita interrupções */

/* Simula o acesso a memória */

  for(wr=0 ; wr < 10 ; wr++);

  disable();                      /* Desabilita interrupções */
  wr = *f & 1;
  wr = *f & 2;
  enable();                       /* Habilita interrupções */
  wr = value | 2;
```

```
 *f = value;
}
#endif


/*************************************************************************
*
* Rotinas para simular as memórias do tipo SAFE
*
*************************************************************************/

#if SAFE

static redirect(i, j, graph)
char *graph;
char i,j;
{
  char *arrow_ij;

   if (i==j) return;

 /* O processo i deve ser sempre menor que j */

   if (i < j) {

     arrow_ij = graph + (i*MAX_PROCS  + j) * sizeof(FORK);
     take_i(arrow_ij);
     offer_i(arrow_ij);
   }
   else {
     arrow_ij = graph + (j*MAX_PROCS + i) * sizeof(FORK);
     take_j(arrow_ij);
     offer_j(arrow_ij);
   }
}


static read_fork(f)
FORK *f;
{
  if (read_k(&f->fi) == read_k(&f->fj)) {          /* if (F == 1)  */
    if (read_k(&f->wqi)) return(EM_USO_I);
                   else  return(OFER_I);
  }
  else {
    if (read_k(&f->wqj)) return(EM_USO_J);
                  else  return(OFER_J);
  }
}

static take_i(f)
FORK *f;
{
   if (read_fork(f) == OFER_J){
     write_k(&f->wpi, TRUE);                       /* wi = TRUE */
     write_k(&f->wqi, TRUE);                       /* F = 1 */
     write_k(&f->fi, f->fj);
```

```
static take_j(f)
FORK *f;
{
   if (read_fork(f) == OFER_I){
     write_k(&f->wpj, TRUE);
     write_k(&f->wqj, TRUE);                          /* wj = TRUE */
     if (read_k(&f->fi) == read_k(&f->fj))
                          write_k(&f->fj, !f->fj);    /* F = 0 */
   }
}


static offer_i(f)
FORK *f;
{
   if (read_fork(f) == EM_USO_I){
      write_k(&f->wpi, FALSE);                        /* wi = FALSE */
        write_k(&f->wqi, FALSE);
   }
}

static offer_j(f)
FORK *f;
{
   if (read_fork(f) == EM_USO_J){
      write_k(&f->wpj, FALSE);                        /* wj = FALSE */
        write_k(&f->wqj, FALSE);
   }
}

static write_k(f, value)
char *f;
char value;
{
 char wr;

/* A função do código abaixo é de igualar o tempo de acesso das
   operações de leitura e escrita. Não tem nenhuma outra função !! */

  disable();                        /* Desabilita interrupções */
  wr = *f & 1;
  wr = *f;
  enable();                          /* Habilita interrupções */

/* Simula o acesso a memória */

  for(wr=0 ; wr < 10 ; wr++);

  disable();                         /* Desabilita interrupções */
  wr = *f & 1;
  wr = *f & 2;
  enable();                          /* Habilita interrupções */

/* Atualiza o bit de concorrência e escreve o valor especificado */

  wr = value | 2;          /* Seta o bit de escrita concorrente */
  *f = wr;
```

```
static read_k(f)
char *f;
{
 extern int random;
 char r1, r2, i, concorrente;

/* Faz a primeira leitura  e reseta o bit de escrita concorrente */

  disable();                        /* Desabilita interrupções */
  r1 = *f & 1;
  *f = r1;
  enable();                          /* Habilita interrupções */

/* Simula o acesso à memória */

  for(i=0 ; i < 10 ; i++);

  disable();                         /* Desabilita interrupções */
  r2 = *f & 1;
  concorrente = *f & 2;
  enable();                          /* Habilita interrupções */

/* Leitura concorrente com escrita (escolhe aleatoriamente 0 ou 1) */

  if (concorrente) return((int)(random & 1));

/* Leitura nao concorrente com escrita  -> r1 = r2 */

  return((int)r1);
}

#endif


/*************************************************************************
*
* Rotinas para simular as memórias do tipo ATÔMICO
*
*************************************************************************/


#if ATOMICO

static redirect(i, j, graph)
char *graph;
char i,j;
{
  char *arrow_ij;

   if (i==j) return;

 /* O processo i deve ser sempre menor que j */

   if (i < j) {
     arrow_ij = graph + i*MAX_PROCS  + j;
     take_i(arrow_ij);
     offer_i(arrow_ij);
   }
```

```
      take_j(arrow_ij);
      offer_j(arrow_ij);
    }
}

static read_fork(fork_ij)
char *fork_ij;
{
 int r;
 char r1;
 extern unsigned int delay;
 unsigned int i;
 extern unsigned long read_total, read_concorrente;

 read_total++;
 r = test_and_set(fork_ij);
 r1 = (char)(r & 3);

/* Simulação do tempo de chaveamento da lógica de arbitragem caso a
   operação seja concorrente */

 if ((r & 0xff00) != 0) {
   read_concorrente++;
   for(i=0; i < 2*delay; i++);
 }

/* Simula o acesso a memória */
 for(i = 0; i < delay; i++);

/* Libera o acesso à célula lida e retorna */

 *fork_ij = r1;
 return((int)r1);
}

static write_fork(f, value)
char *f;
char value;
{
 extern unsigned int delay;
 unsigned int i, st;
 extern unsigned long write_total, write_concorrente;

/* Obtém o acesso à célula desejada */
 write_total++;
 st = test_and_set(f);

/* Simulação do tempo de chaveamento da lógica de arbitragem caso a
   operação seja concorrente */

 if ((st & 0xff00) != 0) {
   write_concorrente++;
   for(i=0; i < 2*delay; i++);
 }

/* Simula o acesso à memória */
 for(i = 0; i < delay; i++);
```

```
}

/* Primitivas de manipulação das estruturas de dados abstratas */

static take_i(fork_ij)
char *fork_ij;
{
   if (read_fork(fork_ij) == OFER_J)
        write_fork(fork_ij, EM_USO_I);
}

static take_j(fork_ij)
char *fork_ij;
{
   if (read_fork(fork_ij) == OFER_I)
        write_fork(fork_ij, EM_USO_J);
}


static offer_i(fork_ij)
char *fork_ij;
{
   if (read_fork(fork_ij) == EM_USO_I)
        write_fork(fork_ij, OFER_I);
}

static offer_j(fork_ij)
char *fork_ij;
{
   if (read_fork(fork_ij) == EM_USO_J)
        write_fork(fork_ij, OFER_J);
}
#endif
```

```
/*************************************************************************
*  PROC0 - Programa do processo 0
*
*************************************************************************/

#include "stdio.h"
#include "procs.h"
#include "stdlib.h"


#define PROC_NUM        0
#define process   proc0

#include "task.h"
```

```
;--------------------------------------------------------------------
; KERNEL.ASM - Este programa implementa um núcleo preemptivo com
;              algoritmo de escalonamento do tipo round-robin.
;
; Autor: Luiz Claudio Maia
;
;--------------------------------------------------------------------

        PAGE    60,132

        TITLE   KERNEL

;------
;       Definição dos segmentos
;------


_TEXT   SEGMENT BYTE PUBLIC 'CODE'
_TEXT   ENDS

_DATA   SEGMENT WORD PUBLIC 'DATA'
_DATA   ENDS

DGROUP  GROUP _DATA
PGROUP  GROUP _TEXT

;------
;       Definições gerais de hardware
;------

        ASSUME DS:DGROUP, CS:PGROUP

CTL_8253        EQU     43H     ; Endereço de IO do 8253 (Timer)
CTL_8259        EQU     20H     ; Endereço de IO do 8259 (Controlador
                                                          de Ints)
TIMER0          EQU     40H     ; Endereço de IO do canal 0 do 8253
EOI             EQU     20H     ; End of interrupt do 8259
MODE            EQU     38H     ; Mode 2 para o canal 0 (Rate generator)
MODE_PC         EQU     36H     ; Modo 3 para o canal 0 (Square wave)

;------
;       Estrutura da TCB
;------

TCB       STRUC

Link      DD      ?
IAX       DW      ?
IBX       DW      ?
ICX       DW      ?
IDX       DW      ?
ISI       DW      ?
IDI       DW      ?
IBP       DW      ?
ISP       DW      ?
ISS       DW      ?
IES       DW      ?
IDS       DW      ?
```

```
ICS         DW      ?

TCB         ENDS

;-------------------------------------------------------------
;
;       Segmento de dados
;
;-------------------------------------------------------------

        EXTRN   _SYS_TSK:FAR  ; Ponteiro para a TCB corrente

        PUBLIC  _RANDOM

_DATA   SEGMENT

        EXTRN  _CRIT_SEC:BYTE, _VIDEO_PT:WORD, _COUNTER:WORD
        EXTRN  _TICKS:WORD, _TICKS2:WORD

LAST_CH         DB      0
_RANDOM         DW      ?

_DATA   ENDS


;-------------------------------------------------------------
;
;       Segmento de código
;
;-------------------------------------------------------------

_TEXT   SEGMENT

;------
;       As variáveis a seguir devem estar no segmento  de  código  por
;       questões de otimização.
;------

SAVINT8         DW      0,0   ; Salva endereço da INT8 original
LEVEL           DB      1     ; Flag da INT8
MASK_INT        DB      1     ; Máscara de interrupção
VIDEO_PT        DW      0     ; Usado para depuração

;--------
;       DSEG - Retorna o segmento de dados (DGROUP)
;-------

        PUBLIC  _DSEG

_DSEG   PROC    FAR

        MOV     AX,DGROUP
        RET

_DSEG   ENDP


;--------
```

```
        PUBLIC  _DISABLE

_DISABLE        PROC    FAR

        CLI             ; Desabilita ints
        RET

_DISABLE        ENDP

;--------
;       ENABLE - Habilita interrupções
;-------

        PUBLIC  _ENABLE

_ENABLE PROC    FAR

        STI             ; Habilita ints
        RET

_ENABLE ENDP

;--------
; TEST_AND_SET -  Implementa  o  test&set  para  simular  o  acesso
;                 simultâneo aos registradores atômicos
;-------

        PUBLIC  _TEST_AND_SET

_TEST_AND_SET   PROC    FAR

        PUSH    BP
        MOV     BP,SP
        PUSH    ES
        PUSH    SI

        LES     SI, DWORD PTR [BP+6]

;------ Testa se esta célula está sendo usada (lida ou escrita)

TS10:
        CLI                             ; Desabilita ints
        CMP     BYTE PTR ES:[SI],3      ; Testa o bit de exclusão...
        JBE     TS20                    ; ... da célula
        STI                             ; Habilita ints.
        JMP     TS10                    ; Tenta novamente

;-----  Seta o bit para indicar o uso da célula

TS20:
        MOV     AL, BYTE PTR ES:[SI]
        OR      BYTE PTR ES:[SI],4
        STI                             ; Habilita ints.

;------ Retorna o valor lido

        POP     SI
```

```
        POP     BP
        RET

_TEST_AND_SET   ENDP


        PUBLIC CLKINT

;------
;       CLKINT
;
;      Esta rotina recebe o controle através da INT08 (interceptada),
;      salva o contexto da tarefa atual e atualiza os registradores
;      para executar uma nova tarefa (A próxima TCB da fila).
;
;------

        ASSUME   DS:DGROUP

        PUBLIC  CLKINT

CLKINT  PROC    FAR


;------ Incrementa o contador de ticks

        PUSH    AX
        PUSH    DS

;------ Envia um EOI para o 8259 e para o contador 0 do 8253

        MOV     AL,EOI
        OUT     CTL_8259,AL         ; Envia um EOI

        MOV     AX,DGROUP           ; Atualiza o segmento de dados
        MOV     DS,AX
        INC     _TICKS2             ; Atualiza o contador de ticks
        JNZ     IC05
        INC     WORD PTR _TICKS2 + 2
IC05:
        POP     DS
        POP     AX

;------ Verifica se o kernel esta sendo executado

        DEC     CS:LEVEL
        JNZ     IC15
        JMP     IC10

;------ Retorna da interrupção
IC15:
        IRET

IC10:
        STI                             ; Habilita as ints.

;------ Salva os registradores da tarefa corrente
```

```
        PUSH    BX                      ; Salva BX

;------ Atualiza DS

        MOV     BX,DGROUP
        MOV     DS,BX                   ; DS = DGROUP

        INC     _RANDOM                 ; Incrementa contador randomico

;------ Incrementa o contador de ticks

        INC     WORD PTR _TICKS
        JNZ     IC25
        INC     WORD PTR _TICKS + 2
IC25:

;------ Armazena os registradores na TCB

        MOV     BX, WORD PTR _SYS_TSK   ;BX = Ponteiro da TCB corrente
        POP     [BX].IBX                ; Armazena BX
        POP     [BX].IDS                ; Armazena DS

;------ Retira IP, CS e Flags da Stack

        POP     [BX].IIP                ; Retira IP da stack
        POP     [BX].ICS                ; Retira CS da stack
        POP     [BX].IFLAGS             ; Retira os flags da stack

;------ Armazena os registradores na TCB

        MOV     [BX].IAX,AX             ; Armazena AX
        MOV     [BX].ICX,CX             ; Armazena CX
        MOV     [BX].IDX,DX             ; Armazena DX
        MOV     [BX].ISI,SI             ; Armazena SI
        MOV     [BX].IDI,DI             ; Armazena DI
        MOV     [BX].IES,ES             ; Armazena ES
        MOV     [BX].IBP,BP             ; Armazena BP
        MOV     [BX].ISP,SP             ; Armazena SP
        MOV     [BX].ISS,SS             ; Armazena SS

;------ Recupera os registradores da próxima tarefa na fila

        MOV     BX,WORD PTR [BX].Link   ; BX aponta para a próxima TCB
        MOV     WORD PTR _SYS_TSK,BX    ; Atualiza ponteiro da TCB ..
                                        ; .. corrente

;------ Atualiza os outros registradores

        MOV     AX,[BX].IAX             ; Atualiza AX
        MOV     CX,[BX].ICX             ; Atualiza CX
        MOV     DX,[BX].IDX             ; Atualiza DX
        MOV     SI,[BX].ISI             ; Atualiza SI
        MOV     DI,[BX].IDI             ; Atualiza DI
        MOV     ES,[BX].IES             ; Atualiza ES
        MOV     BP,[BX].IBP             ; Atualiza BP

;------ Troca para a stack do novo processo
```

```
        MOV     SP,[BX].ISP             ; Atualiza SP
        MOV     SS,[BX].ISS             ; Atualiza SS

;------ Recoloca novos CS, IP e Flags na nova Stack

        PUSH    [BX].IFLAGS             ; Recoloca os Flags na stack
        PUSH    [BX].ICS                ; Recoloca CS na stack
        PUSH    [BX].IIP                ; Recoloca IP na stack

        PUSH    [BX].IDS                ; Atualiza BX
        MOV     BX,[BX].IBX

;------ Habilita a interrupção do 8253

        PUSH    AX
        MOV     AL,MODE                 ; Para o contador 0 do 8253
        OUT     CTL_8253,AL
        POP     AX

;------ Recomeca a contar o tempo

        CALL    START_COUNTER           ; Carrega o contador 0 do 8253
        POP     DS                      ; Atualiza DS

        IRET

CLKINT  ENDP


;------
;       INSTALL
;
;       Instala o kernel na interrupção de relógio  (INT 8)
;------

        ASSUME   DS:DGROUP, ES:_TEXT

        PUBLIC  _INSTALL

_INSTALL        PROC    FAR

;------ Salva os registradores

        PUSH    DS                      ; Salva DS
        PUSH    ES                      ; Salva ES

;------ Inicializa ES

        MOV     AX,_TEXT                ; Segmento de código
        MOV     ES,AX                   ; Inicializa ES

        CLI                             ; Desabilita interrupções

;------ Salva interrupção do Timer (8H)

        MOV     DX,ES                   ; Salva ES
        MOV     AL,8                    ; Interrupção do Timer
```

```
        MOV     AX,ES                   ; Salva o segmento
        MOV     ES,DX                   ; Restaura ES
        MOV     ES:SAVINT8,BX           ; Guarda o offset
        MOV     ES:SAVINT8+2,AX         ; Guarda o segmento
        PUSH    DS                      ; Salva DS
        MOV     AX,ES                   ; Segmento de código residente
        MOV     DS,AX                   ; Inicializa DS
        MOV     DX,OFFSET _TEXT:CLKINT  ; Aponta DX para rotina de int.
        MOV     AL,8                    ; Interrupção do Timer
        MOV     AH,25H                  ; Função: Set Interrupt Vector
        INT     21H
        POP     DS                      ; Restaura DS

;------ Altera a programação do 8253 para obter um tempo de chaveamento
;       menor

        MOV     AX,DGROUP
        MOV     DS,AX                   ; DS = DGROUP

        MOV     AX,MODE                 ; Seleciona o modo 2 de
        OUT     CTL_8253,AL             ; .. operação ara o 8253

        CALL    START_COUNTER           ; Carrega o contador 0 do 8253

;------ Restaura os registradores e retorna a quem chamou

        STI                             ; Habilita interrupções
        POP     ES                      ; Restaura ES
        POP     DS                      ; Restaura DS
        RET                             ; Retorna a quem chamou

_INSTALL        ENDP


START_COUNTER   PROC    NEAR

        PUSH    AX                      ; Salva AX

        MOV     AL, BYTE PTR _COUNTER   ; Nova freqüência especificada
        JMP     I20                     ; Espera um tempo
I20:    JMP     I21
I21:    JMP     I22
I22:
        OUT     TIMER0,AL               ; Atualiza LSB do divisor
        MOV     AL, BYTE PTR _COUNTER+1
        JMP     I30                     ; Espera um tempo
I30:    JMP     I31
I31:    JMP     I32
I32:
        OUT     TIMER0,AL               ; Atualiza o MSB do divisor

        POP     AX                      ; Recupera AX
        RET

START_COUNTER   ENDP
```

```
;       REMOVE
;
;       Remove o kernel na interrupção de relógio
;------

        PUBLIC  _RETIRA

_RETIRA PROC    FAR

;------ Salva os registradores

        PUSH    ES                      ; Salva ES

;------ Inicializa ES

        MOV     AX,_TEXT                ; Segmento de código
        MOV     ES,AX                   ; Inicializa ES
        CLI                             ; Desabilita interrupções

;------ Restaura a int 8 original

        PUSH    DS                      ; Salva DS
        MOV     AX,ES:SAVINT8+2         ; Segmento de código residente
        MOV     DS,AX                   ; Inicializa DS
        MOV      DX,ES:SAVINT8          ; Aponta DX para rotina de int.
        MOV     AL,8                    ; Interrupção do Timer
        MOV     AH,25H                  ; Função: Set Interrupt Vector
        INT     21H
        POP     DS                      ; Restaura DS

;------ Retorna a programação original do 8253

        MOV     AX,MODE_PC
        JMP     R10                     ; Espera um tempo
R10:    JMP     R11
R11:    JMP     R12
R12:
        OUT     CTL_8253,AL
        MOV     AL, 0                   ; Retorna ao contador original
        JMP     R20                     ; Espera um tempo
R20:    JMP     R21
R21:    JMP     R22
R22:
        OUT     TIMER0,AL               ; LSB = 0  contador = 65536
        JMP     R30                     ; Espera um tempo
R30:    JMP     R31
R31:    JMP     R32
R32:
        OUT     TIMER0,AL               ; MSB = 0


;------ Reseta o teclado

        IN      AL,61H
        MOV     AH,AL
        OR      AL,80H
        OUT     61H,AL
        JMP     R40
```

```
        OUT     61H,AL
        STI                             ; Habilita interrupções

;------ Envia um EOI para o 8259

        MOV     AL,EOI
        OUT     CTL_8259,AL             ; Envia um EOI

;------ Restaura os registradores e retorna a quem chamou

        POP     ES                      ; Restaura ES
        RET                             ; Retorna a quem chamou

_RETIRA ENDP

_TEXT   ENDS

        END     _TEXT:_INSTALL
```

Referências Bibliográficas

[1] DOLEV, D., GAFNI, E. e SHAVIT, N., "Toward a Non-Atomic Era: ℓ-Exclusion as a Test Case", ACM Symposium on Theory of Computing, pp. 78-92, 1988.

[2] LAMPORT, L., "On Interprocess Communication. Part I: Basic Formalism", *Distributed Computing*, vol. 1, nº 2, pp. 77-85, 1986.

[3] LAMPORT, L., "On Interprocess Communication. Part II: Algorithms", *Distributed Computing*, vol. 1, nº 2, pp. 86-101, 1986.

[4] LAMPORT, L., "The Mutual Exclusion Problem. Part I: A Theory of Interprocess Communication", *Journal of ACM*, vol. 33, nº 2, pp. 313-326, 1986.

[5] LAMPORT, L., "The Mutual Exclusion Problem. Part II: Statements and Solutions", *Journal of ACM*, vol. 33, nº 2, pp. 327-348, 1986.

[6] FISCHER, M. J., LYNCH, N. A., BURNS, J. E., e BORODIN, A., "Resource Allocation with Immunity to Limited Process Failure", *Proc. 20th IEEE Symp. on Foundations of Computer Science*, pp. 234-254, 1979.

[7] FISCHER, M. J., LYNCH, N. A., BURNS, J. E., e BORODIN, A., "Distributed Fifo Allocation of Identical Resources Using Small Shared Space", *MIT/LCS/TM-290*, 1985.

[8] DIJKSTRA, E. W., "Solution of a problem in concurrent programing control", *Communication of ACM*, vol. 8, nº 9, pp. 569, 1965.

[9] CHANDY, K. M., MISRA, J., "The Drinking Philosophers Problem", *ACM Transaction on Programming Languages and Systems*", vol. 6, no 4, pp. 632-646, 1984.