Aula 14

José A. Cardoso e Cunha
DI-FCT/UNL

Este texto resume o conteúdo da aula teórica.

1 Objectivo

Objectivo da aula: Problemas de interferência entre processos concorrentes. Exemplos. Problema de exclusão mútua e regiões críticas. Classificação de soluções e suas limitações: inibição das interrupções de programa, algoritmo de Dekker, instrução TEST-AND-SET

2 Interferências entre processos concorrentes

Um dos principais problemas da programação concorrente é a possibilidade de os erros de programação terem manifestações dependentes do tempo, isto é, visíveis numas execuções e não visíveis noutras. Por exemplo, num SO com multiprogramação, mesmo que só haja um CPU, existem múltiplos processos concorrentes em diversas etapas da sua execução, seja associados a múltiplos utilizadores interactivos, seja associados ao controlo de múltiplos dispositivos periféricos, seja associados a tarefas internas de gestão dos recursos controlados pelo SO. A imprevisibilidade da ocorrência das interrupções de programa, e a aleatoriedade dos momentos em que um processo pode perder o controlo do CPU, para se bloquear, ou para dar a vez a outros processos, exige que o programador tenha especial cuidado, quando existem zonas do programa nas quais se acedem a estruturas de dados em memória partilhada.

3 Exemplo 1

A figura 1 ilustra uma situação em que dois processos concorrents, P1 e P2, acedem a uma variável **x** em memória partilhada, sobre a qual efectuam a operação *inc x*, para incrementar o valor de x de uma unidade. Contudo,

admitimos, neste exemplo, que não existe uma instrução máquina capaz de efectuar o incremento do conteúdo de uma célula de memória, pelo que esta acção é decomposta nas seguintes acções elementares (isto é, cada uma destas corresponde a uma instrução máquina), em que Mem[x] representa a célula de endereço correspondente à variável x, e r1 representa um registador do CPU:

```
r1 := Mem[x];
r1 := r1 + 1;
Mem[x] := r1;
```

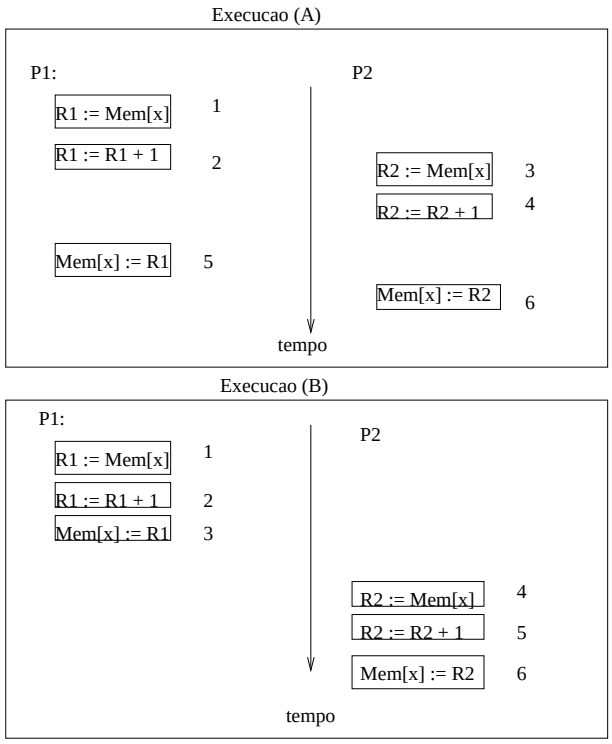

Figura 1: Execuções concorrentes.

Na figura, a numeração indica as ordens de execução das instruções num sistema de CPU com um SO com multiprogramção. Se, no início, x valer 1,

então, no fim da execução (B), x valerá 3, mas, no fim da execução (A), x valerá 2, o que significa que se perdeu uma das actualizações.

Note-se que estas são apenas duas das muitas ordens de execução possíveis, pelo que, em geral, se torna impossível fazer uma análise exaustiva de todos as sequências de execução possíveis, para verificar em quais poderá haver manifestações de erros.

Neste exemplo, uma solução, aparentemente trivial, seria admitir, ou melhor, exigir, que houvesse uma instrução máquina para a acção *inc x*.

4 Exemplo 2

Considere as figuras 2, 3, 4, 5 e 6. Na parte (a)

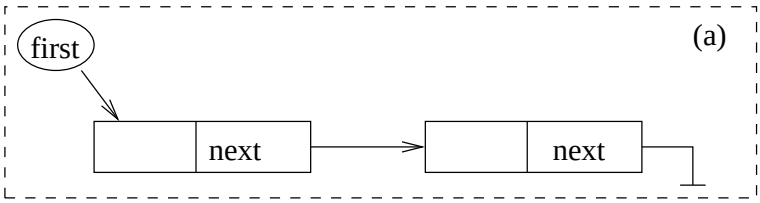

Figura 2: Execuções concorrentes.

mostra-se uma lista ligada de elementos. Na parte (b),

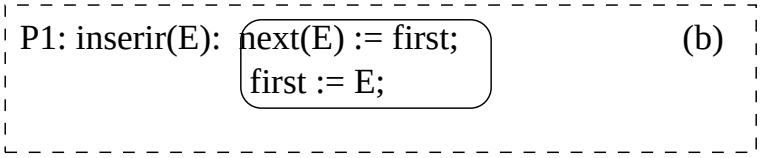

Figura 3: Execuções concorrentes.

o pseudo-código da operação inserir(E), com o novo elemento e inserido à cabeça, na parte (c).

Na parte (d) mostra-se um cenário possível da execução concorrente de duas operações inserir(E) e inserir(F), invocadas pelos processos P1 e P2, sendo indicada a ordem de execução de cada acção, pela numeração de 1 a 4.

Na parte (e) mostra-se o resultado final, vendo-se como o elemento F foi perdido. Note-se que, noutros cenários de execução, o elemento F pode não

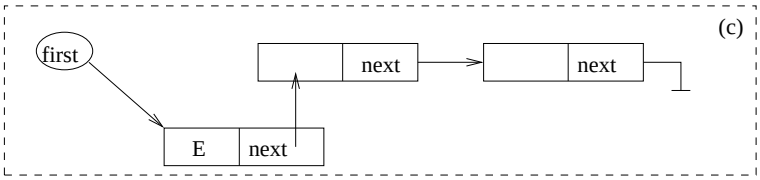

Figura 4: Execuções concorrentes.

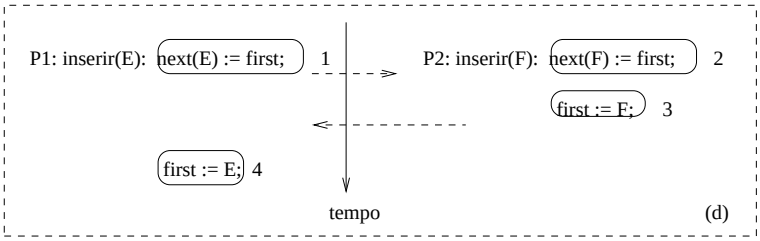

Figura 5: Execuções concorrentes.

ser perdido!

5 Controlo do acesso aos recursos partilhados

Certos recursos são privados a cada processo: o seu próprio código, as zonas de dados e de pilha. No entanto, se utilizarmos as chamadas ao SO, como no Unix *shmop*, estaremos a definir regiões de dados que vão ser partilhadas

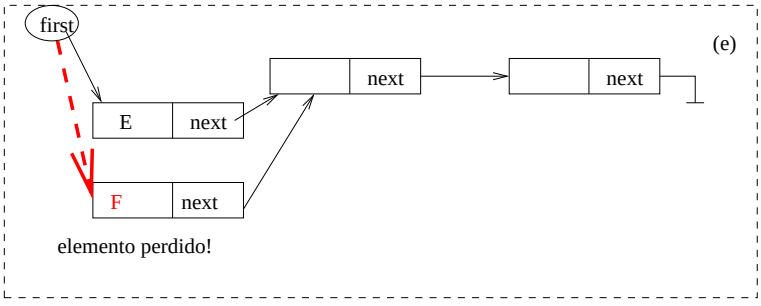

Figura 6: Execuções concorrentes.

por múltiplos processos.

Nesse caso, dependendo do tipo de acessos que façamos, essas regiões partilhadas poderão ter de serem consideradas como *recursos críticos*. Quando há um recurso crítico, o código que, em cada processo, lhe dá acesso, chama-se uma *secção* ou *região crítica* (figura 7).

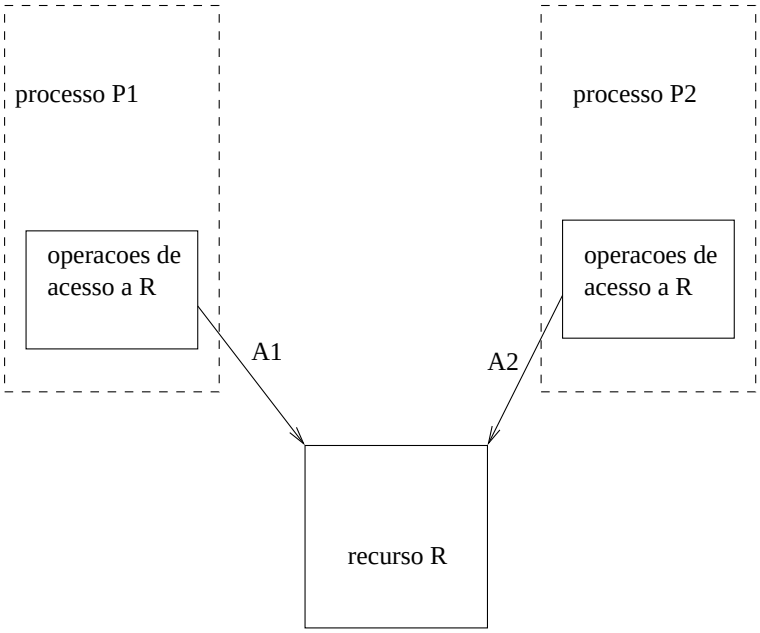

Figura 7: Regiões críticas.

Se R é um recurso crítico, então as acções A1 e A2 devem excluir-se mutuamente, de tal forma que, a cada momento, apenas um processo esteja a executar a sua respectiva região crítica. A figura 8 mostra a ideia básica para realizar regiões críticas. É preciso que, em cada processo, a entrada na sua região crítica seja precedida de um protocolo no qual se garanta que, em caso de o recurso não estar a ser acedido no momento, este processo será o único a conseguir iniciar a sua região crítica e que, em caso de o recurso estar a ser acedido por outro processo, todos os outros terão de aguardar até que esse processo acabe a sua região crítica.

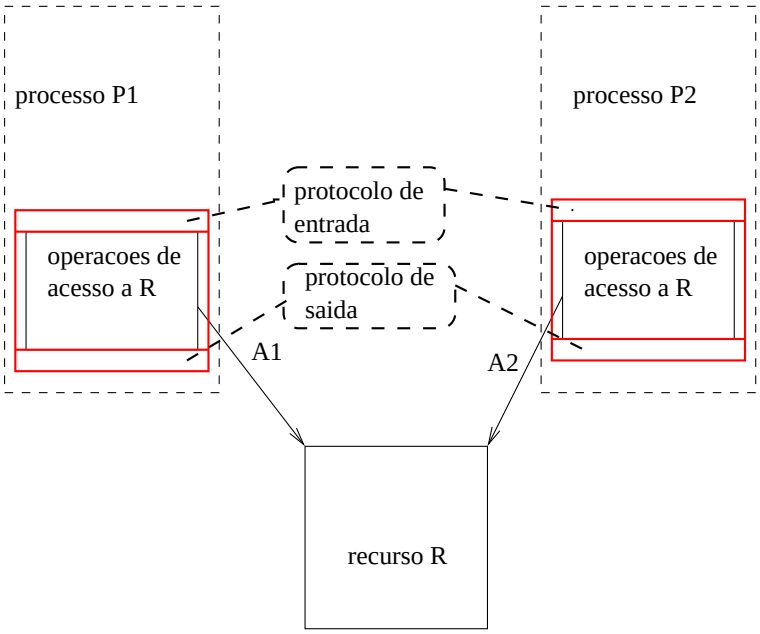

Figura 8: Protocolos de acesso a regiões críticas.

6 Implementação de regiões críticas

A implementação dos protocolos de entrada nas regiões críticas dependem de várias dimensões que caracterizam o ambiente de execução de programas:

- um CPU ou múltiplos CPU
- execução em modo utilizador ou modo supervisor
- sistema de controlo em tempo real ou não

6.1 Sistemas com um só CPU

Se o sistema só tem um CPU e os programas são executados sob o controlo de um sistema de multiprogramação, a única maneira de um processo perder o controlo do CPU é, como se sabe, através do mecanismo de interrupções:

- ou porque fez uma chamada ao SO: *supervisor call* gera uma interrupção por software, na qual se bloqueou, eg à espera de dados;

- ou porque foi interrompido pelo TIME-SLICE;

Em qualquer dos casos, se for possível inibir o atendimento de interrupções durante a região crítica, através da instrução *disable interrupts*, estaria o problema resolvido? A resposta é sim, desde que não seja grave deixar de atender interrupções durante a região crítica. Se esta for muito longa e o sistema tiver dispositivos periféricos que exijam resposta em tempo 'útil', pode não ser aceitável ter o atendimento das interrupções inibido.

Também não é aceitável utilizar este método se houver o risco de um programa, por erro ou por outra razão, ficar por exemplo, envolvido num ciclo 'eterno' dentro da sua região crítica. Se as interrupções estão inibidas, nem o mecanismo de TIME-SLICE o consegue tirar de lá!

Finalmente, se os programas que incluem as regiões críticas são executados em *modo utilizador* (o que é o modo normal pois é o que garante a protecção dos mapas de memória dos processos e garante ao SO o controlo total das operações), então o programa utilizador não tem acesso à instrução máquina *interrupt disable*, donde este método não pode ser aplicado!

Assim, em conclusão, a única situação em que o método de inibir as interrupções pode ser aplicado para realizar regiões críticas é:

- se a execução for em *modo supervisor*, por exemplo, para os programas do núcleo do próprio SO;
- se não houver problemas de resposta em tempo real, ou seja se a duração da região crítica for o suficientemente pequena, para não perturbar o atraso na resposta;
- se o código da região crítica tiver a garantia de que não tem erros, do tipo dos ciclos eternos.

Uma alternativa, em geral utilizada apenas em ambientes dedicados ao controlo de tarefas, é, em vez de inibir as interrupções, inibir a comutação de processos, no núcleo de multiprogramação. A ideia é permitir a ocorrência de interrupções, mas dar a garantia que o processo que é interrompido no meio da região crítica, tem a certeza de que, após a interrupção tratada, volta a receber o controlo de imediato, sem que qualquer outro processo possa executar. Claro que isto só funciona se a própria rotina de serviço de interrupções não for aceder ao recurso crítico! De qualquer forma, esta alternativa, que exige acesso a uma primitiva do núcleo do tipo *inibir escalonador de processos*, apenas é aplicável quando o ambiente de execução é dedicado ao controlo de dispositivos hardware específicos. Não é uma solução aplicável

num sistema de multiprogramação genérico, como o Unix, em que o programa utilizador não tem acesso aos mecanismos de escalonamento de processos do núcleo.

6.2 Sistemas com múltiplos CPU

Neste caso, inibir interrupções não resolve o problema, pois só tem efeito sobre um determinado CPU e não sobre os outros. Nada impediria os programas em execução em outros CPU de acederem simlutaneamente à memória partilhada através do bus comum. O mecanismo de interrupções não impede os programas de acederem à memória partilhada num sistema de múltiplos processadores.

6.3 Soluções genéricas

Pela discussão anterior, se conclui serem necessárias soluções genéricas para implementar o protocolo de entrada nas regiões críticas. Estas soluções não devem ser baseadas na inibição do mecanismo de interrupções. A solução genérica é, conceptualmente, muito simples (9).

O valor booleano R indica se o recurso está correntemente a ser acedido ou não. Ao executar o protocolo de entrada, cada processo deve testar o valor de R e, caso esteja FALSE, deve afectá-lo a TRUE e pode entrar na sua região crítica. Caso contrário, deverá aguardar, até que encontre o valor R a FALSE. O problema desta solução é que exige precauções para evitar que dois processos concorrentes, encontrando simultaneamente o valor de R a FALSE, tentem ambos afectá-lo a TRUE e entrem ambos nas suas regiões críticas.

7 Tentativa de solução 1

A figura 10 mostra esta primeira tentativa de solução, infelizmente incorrecta!

Ambos os processos podem encontrar a variável partilhada LIVRE a TRUE e afectá-la a FALSE, entrando na região crítica. Para isso, basta que, no início, quando LIVRE está a TRUE, um dos processos perca o controlo do CPU, logo após ter testado o valor de LIVRE, no WHILE, mas imediatamente antes de ter conseguido afectar-lhe FALSE.

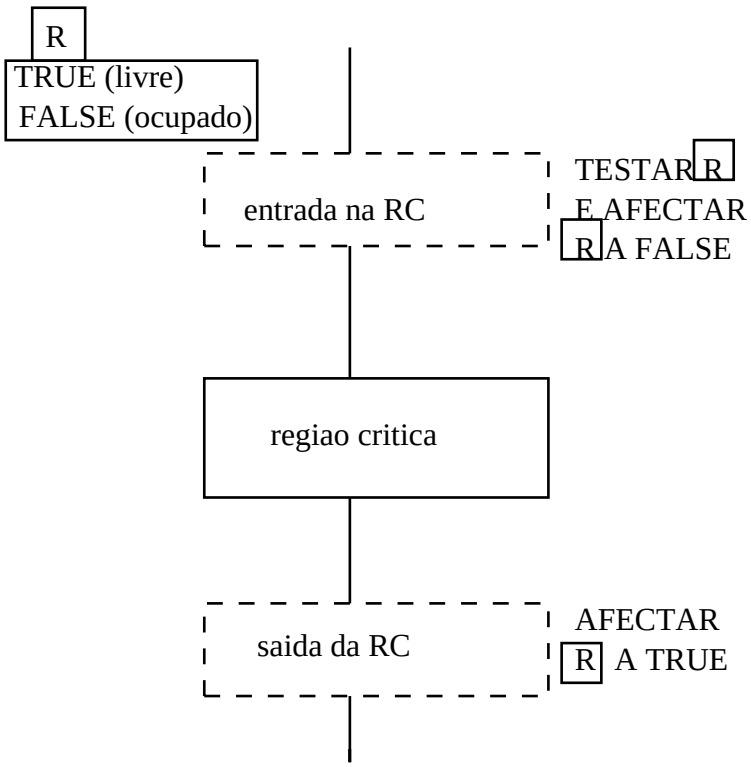

Figura 9: Solução genérica.

8 Tentativa de solução 2

A figura 11 mostra uma segunda tentativa de solução, também infelizmente incorrecta!

Se ambos os processos fazem C1 := 0 e C2 := 0, podem depois encontrar os valores correspondentes do outro processo também a 0, ficando envolvidos num bloqueio eterno, pois não conseguem sair dos ciclos WHILE. Isto pode acontecer, se o primeiro processo, por exemplo, tendo feito C1 := 0, perde o controlo do CPU, permitindo que o outro processo também faça C2 := 0.

Uma boa solução não deve permitir situações dependentes do ritmo de execução relativo dos processos concorrentes.

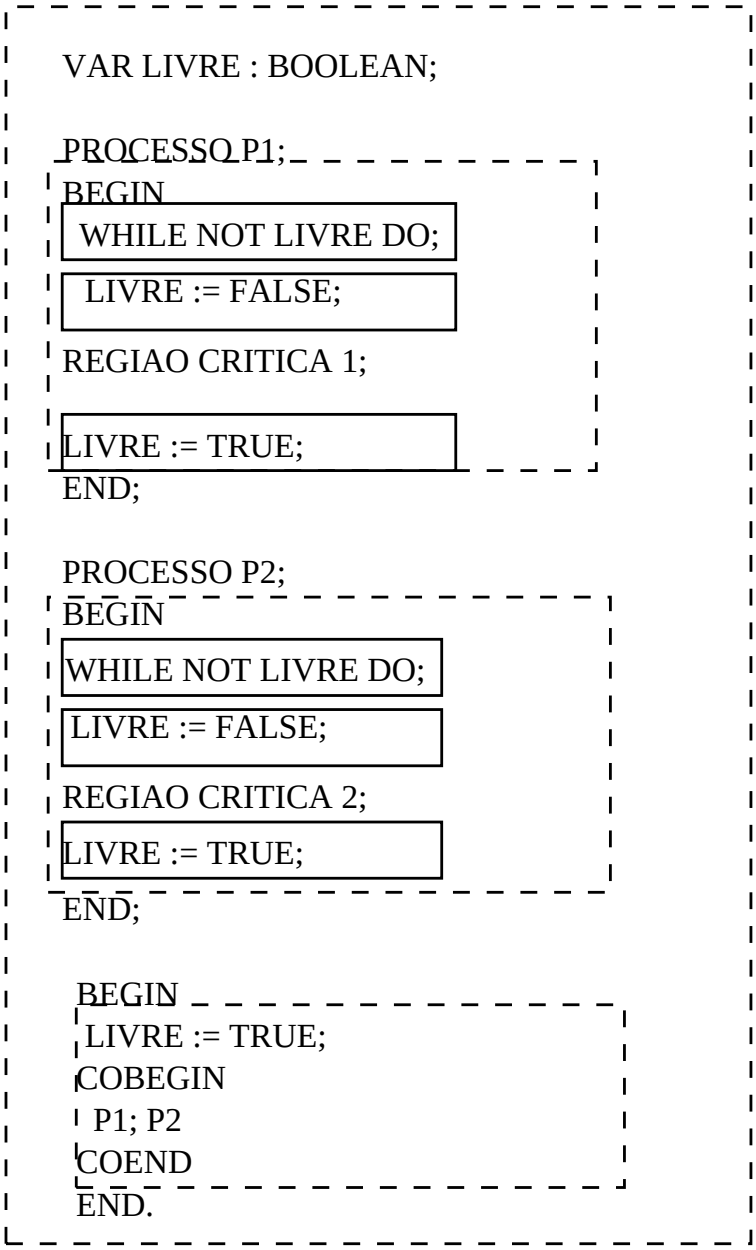

Figura 10: Tentativa 1.

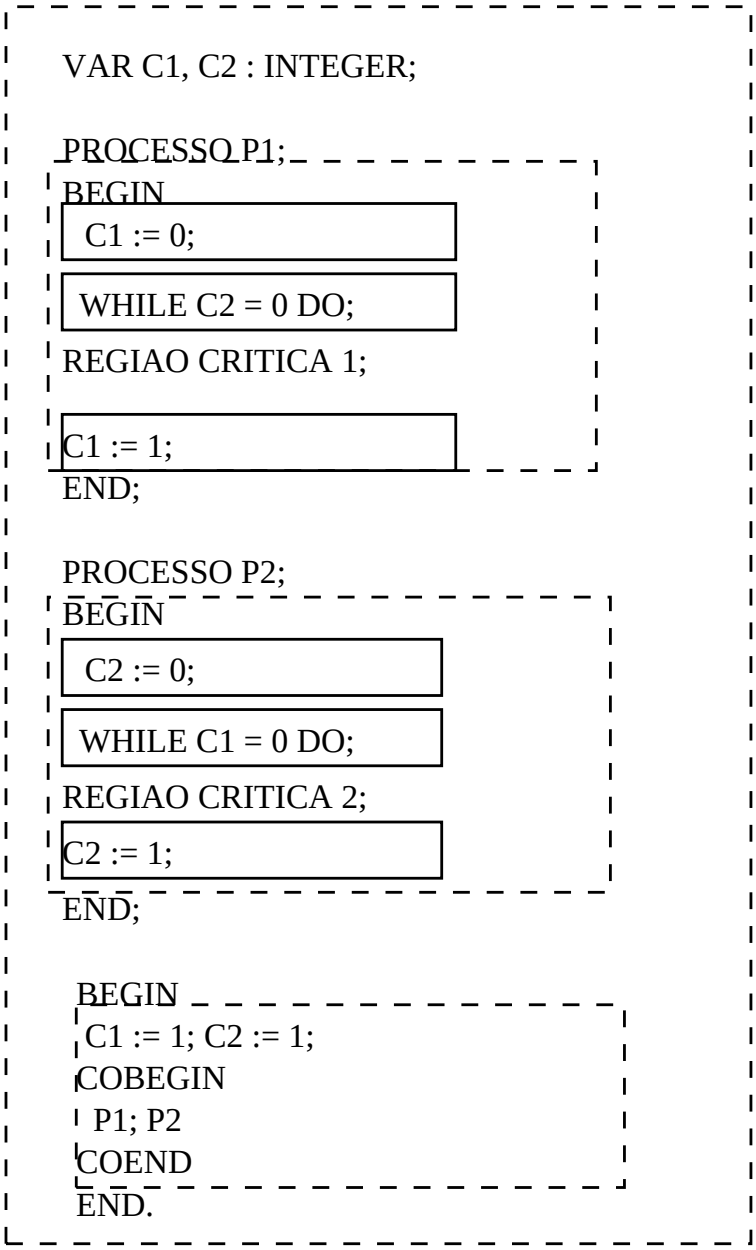

Figura 11: Tentativa 2.

9 Tentativa de solução 3

A figura 12 mostra uma terceira tentativa de solução, também infelizmente incorrecta!

A solução é correcta, do ponto de vista da garantia de que, a cada momento, só há um processo a executar a sua região crítica.

Contudo, no caso frequente em que os processos podem precisar de aceder múltiplas vezes às regiões críticas (por exemplo, considere que, no programa indicado, cada processo tinha um ciclo mais envolvente, de tipo WHILE true DO , que o transformava num processo cíclico eterno - como um servidor), a solução exige uma disciplina muito rígida de execução, forçosamente alternada entre P1 e P2, isto é, as sequências possíveis de execução são sempre da forma: P1/P2/P1/P2/.... Isto não é aceitável porque exige que os processos estejam fortemente relacionados entre si. Este estilo de execução é característico do das *co-rotinas*, um modelo de execução concorrente em que várias rotinas vão passando o controlo de umas para as outras.

Em geral, para processos independentes e não relacionados, isto não funciona bem. Por exemplo, imagine que o processo P2, por erro ou por terminação normal, acaba. A partir daí, admitindo que a variável vez ficou igual a 1, o processo P1 pode aceder mais uma vez à sua região crítica, mas depois deixa vez = 2 e, como o P2 já terminou, P1 nunca mais pode voltar a entrar.

10 Tentativa de solução 4

A figura 13 mostra uma quarta tentativa de solução, também infelizmente incorrecta!

Cada processo (e.g. P1), ao entrar no ciclo WHILE, após ter verificado que o outro (P2, com c2 = 0) também estava a tentar entrar ou já lá estava dentro, desiste 'temporariamente' (põe c1 = 1) e depois, possivelmente após algum tempo, volta a insistir (põe c1 = 0). Em princípio, verificar-se-á uma probabilidade muito pequena de que, estando ambos a tentarem entrar concorrentemente, os dois desistam e insistam a um ritmo tal que encontram sempre o outro a insistir e, assim, nunca saiam dos seus ciclos WHILE. Contudo, essa possibilidade existe e conduziria a uma situação de bloqueio mútuo dos dois processos. Assim, esta tentativa deve também ser rejeitada, pois uma solução do problema tem de funcionar em todos os possíveis cenários de execução.

Apesar de tudo, esta tentativa dá uma boa ideia para se resolver este

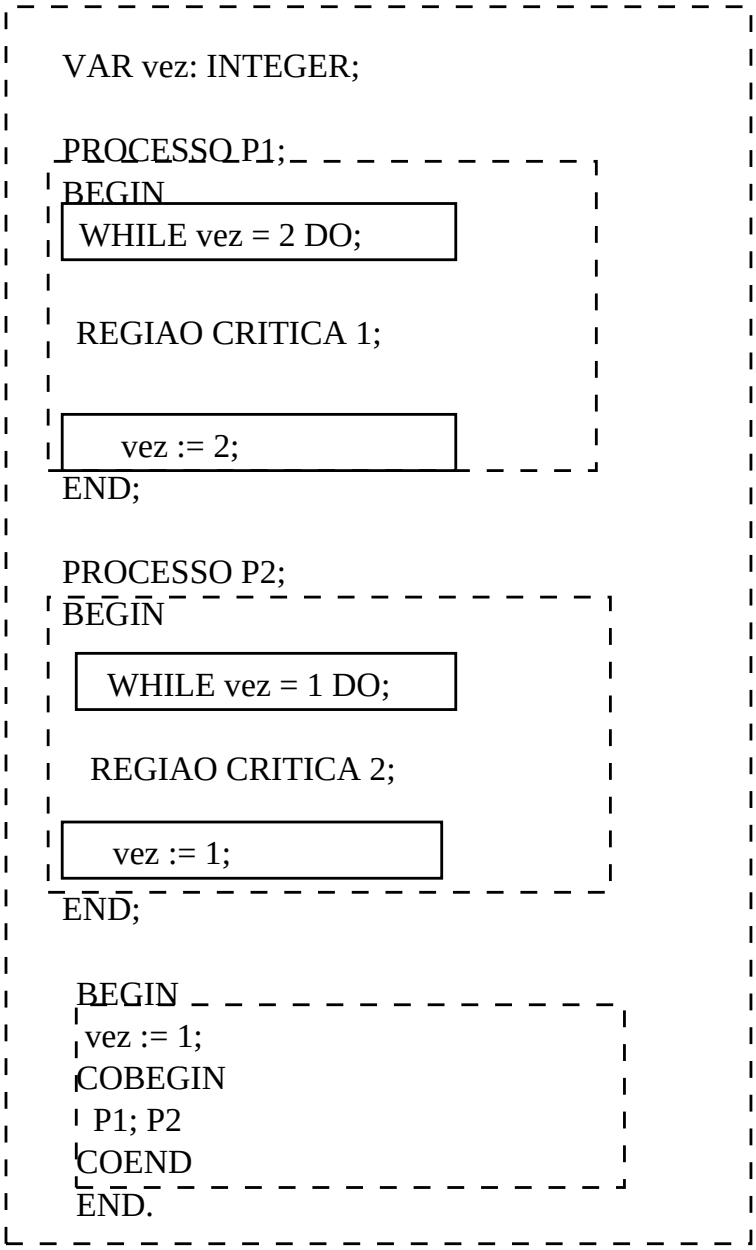

Figura 12: Tentativa 3.

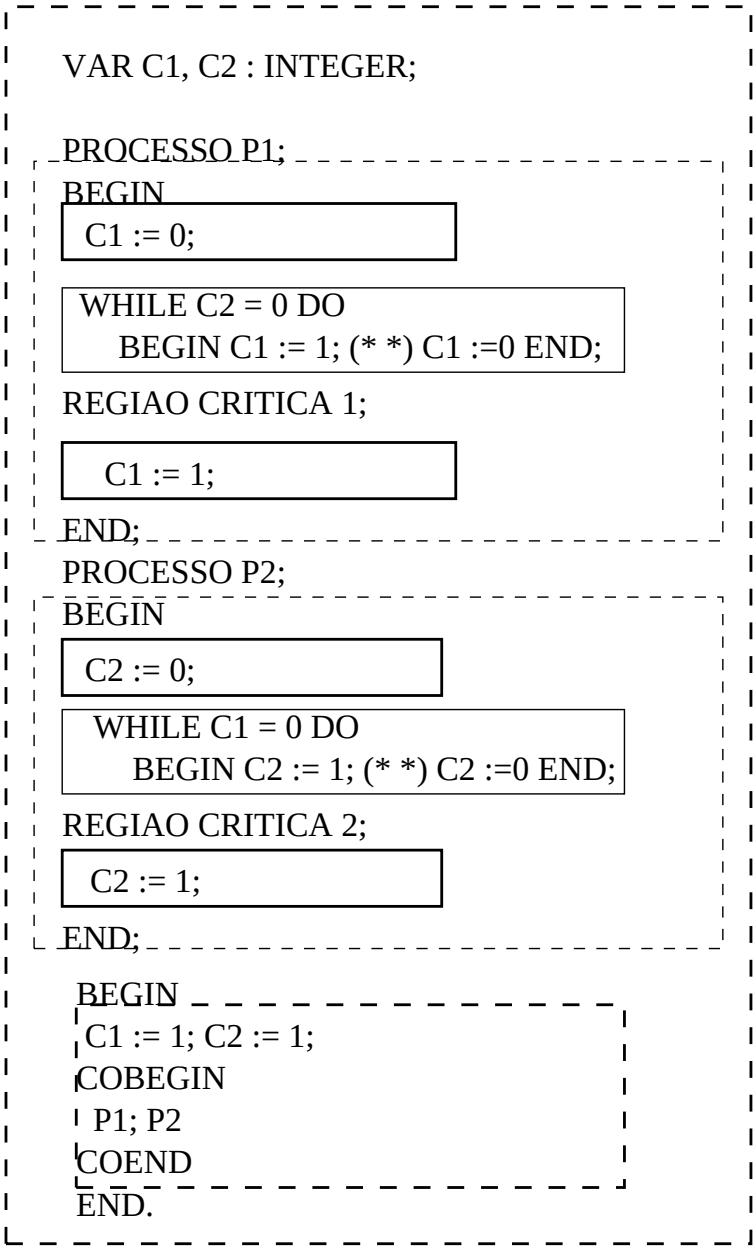

Figura 13: Tentativa 4

problema. Basta tornar o comportamento dos dois processos assimétrico, isto é, em vez de desistirem e insistirem ambos, obrigamos apenas um deles a desistir e mantêmo-lo desistindo, enquanto o outro não tiver saído da região crítica.

11 Algoritmo de Dekker

A figura 14 mostra uma solução correcta, devida a Dekker, que segue a sugestão antes apontada.

12 Conclusões

O Algoritmo de Dekker resolve o problema da *exclusão mútua*, que foi ilustrado para o caso de dois processos concorrentes, partilhando um recurso num sistema de memória partilhada. O algoritmo utiliza uma abordagem baseada em espera activa (*busy activa*), na qual um processo, que não pode aceder à sua região crítica, fica em ciclo de teste de variáveis partilhadas.

Uma solução deste tipo só é aceitável se houver um número de CPUs igual (ou superior) ao número de processos que se devam executar no sistema. Porquê?

Porque, nesse caso, podemo-nos dar 'ao luxo' de consumir ciclos de CPU, processos que não fazem mais nada senão aguardar até que possam entrar. Como esta situação é rara na prática, onde, em geral, há mais processos do que CPUs disponíveis – em geral, na maioria dos computadores, ainda só há um único CPU –, isto significa que a solução de Dekker é pouco usada.

Contudo, a solução de Dekker mostrou que é possível resolver o problema inteiramente por *software*, isto é, sem exigir instruções máquina especiais.

A solução de Dekker tem outra desvantagem, igualmente desagradável. Não permite uma expansão de escala, em termos do número de processos participantes no algoritmo, sem que se faça uma modificação significativa do programa que realiza o algoritmo. Se quisermos estender aquela solução, de 2 para N processos, teremos de ter:

- alteração do cobegin/coend, para criar N processos'
- declaração de um vector de N variáveis Ci;
- assumir que a variável *vez* passa a valer de 1 a N;

- alteração das condições de teste e afectação das variáveis partilhadas em **todos** os processos envolvidos: os 2 existentes, mais os N-2 novos processos, que queríamos introduzir no algoritmo.

Esta falta de modularidade não é aceitável em sistemas onde haja criação dinâmica de processos. Precisamos de outras soluções, nas quais a entrada de um novo processo no sistema não implique a reescrita total do código de todos os outros processos (e a saída de cada processo, também!).

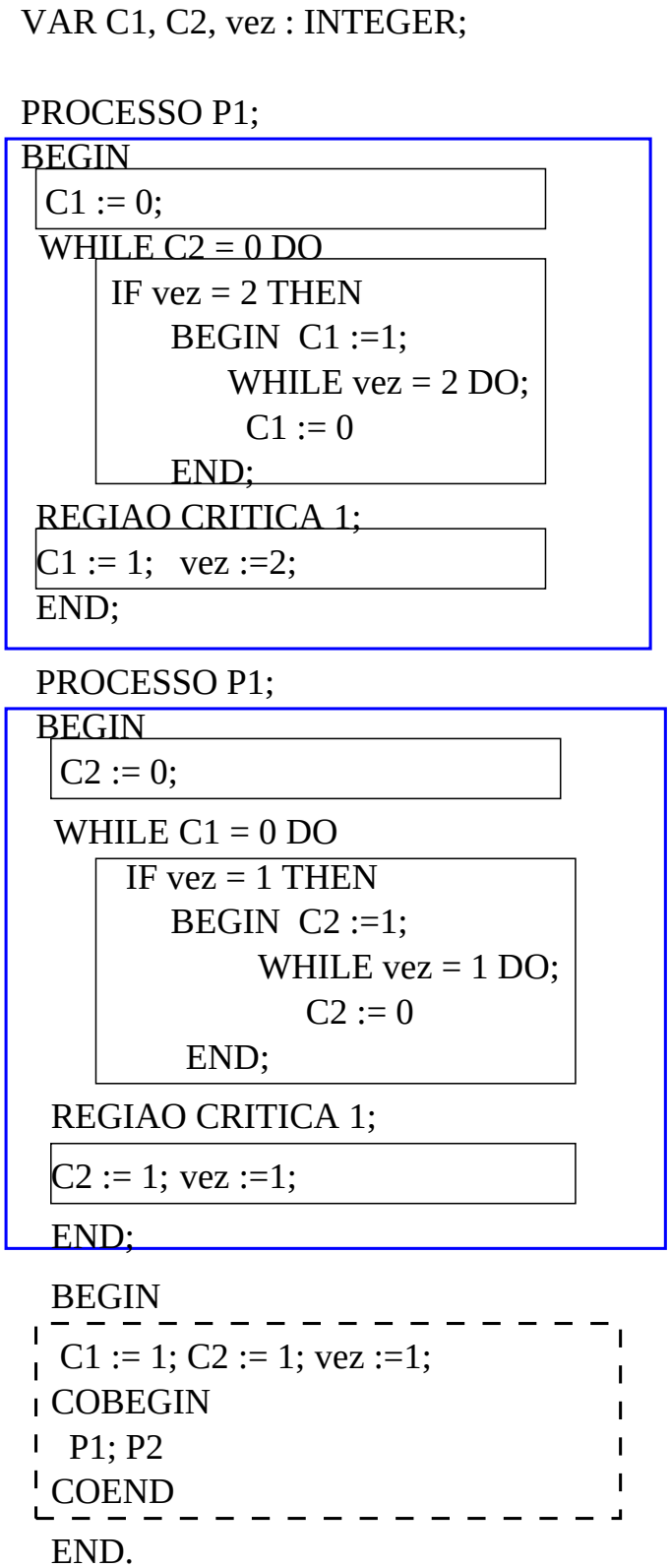

Figura 14: Algoritmo de Dekker.