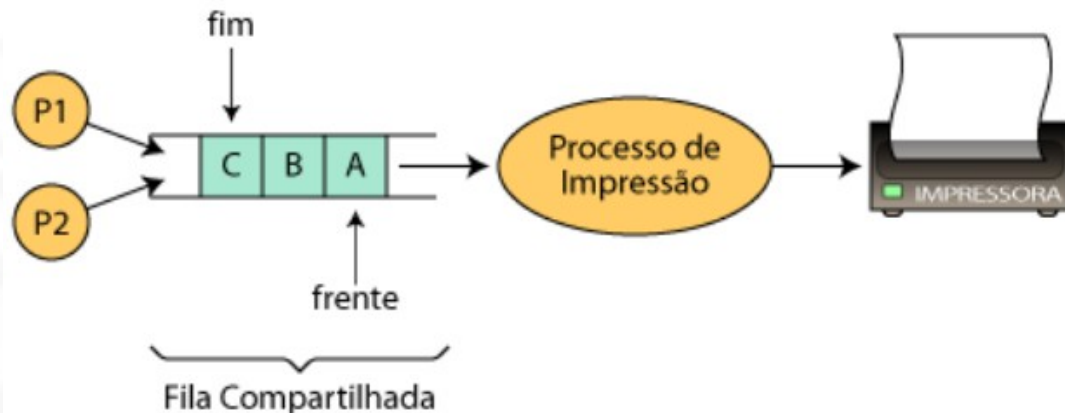


# Sincronização de Processos (1)

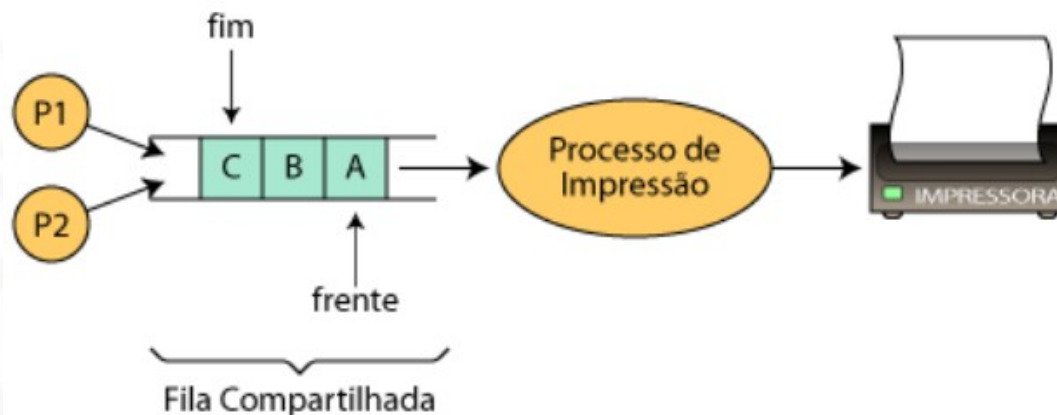
Mecanismos de busy wait

# Condições de Corrida



- Exemplo: Fila de impressão.
  - Qualquer processo que queira imprimir precisa colocar o seu documento na fila de impressão (compartilhada).
  - O processo de impressão retira os documentos na ordem em que chegaram na fila
  - Se a fila é compartilhada, isto significa que seus dados, assim como os indicadores de **frente** e **fim** da fila também o são

# Condições de Corrida



1.  $fim++$  (incrementa o indicador do fim da fila)
  2. coloca documento na posição do novo fim da fila
- dois processos resolvem simultaneamente imprimir um documento
  - o primeiro processo foi interrompido (por ter acabado o seu quantum) entre os comandos 1 e 2
  - o segundo processo insere seu documento na fila antes que o primeiro processo tenha acabado : **qual é o erro ????**
  - Há uma **condição de corrida** quando dois ou mais processos estão acessando dados compartilhados e o resultado depende de quem roda quando

## Outro exemplo ...

```
void Procedure echo();  
var   chin, chout: character; //compartilhadas  
begin  
    input (chin, keyboard);  
    chout := chin;  
    output (chout, display);  
end.
```

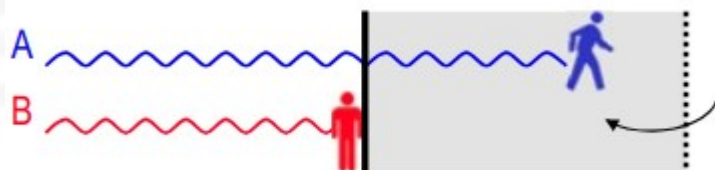
- P1 invoca *echo()* e é interrompido imediatamente após a conclusão da função *input()*. Suponha que *x* tenha sido o caractere digitado, que agora está armazenado na variável *in*.
- P2 é despachado e também invoca *echo()*. Suponha que *y* seja digitado (*in* recebe *y*), sendo então exibido no dispositivo de saída.
- P1 retoma a posse do processador. O caractere exibido não é o que foi digitado (*x*), pois ele foi sobreposto por *y* na execução do processo P2. Conclusão: o caractere *y* é exibido duas vezes.
- Essência do problema: **o compartilhamento da variável global *in*.**

– **Condição de corrida !**

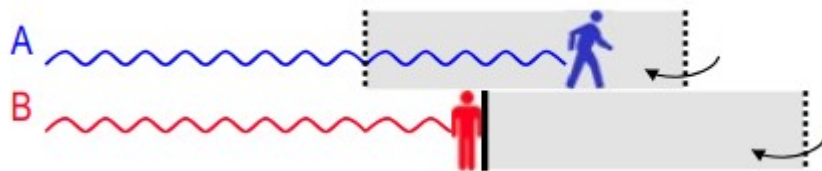
# Condições de Corrida

- **Condições de corrida** são situações em que dois ou mais processos acessam dados compartilhados e o resultado final depende da ordem em que os processos são executados
  - Ordem de execução é ditada pelo mecanismo de escalonamento do S.O.
  - Torna a depuração difícil.
- Condições de corrida são evitadas por meio da introdução de mecanismos de **exclusão mútua**:
  - A exclusão mútua garante que somente um processo estará usando os dados compartilhados num dado momento.
- **Região Crítica**: parte do programa (trecho de código) em que os dados compartilhados são acessados
- Objetivo da Exclusão Mútua:
  - Proibir que mais de um processo entre em sua Região Crítica

# Exclusão Mútua

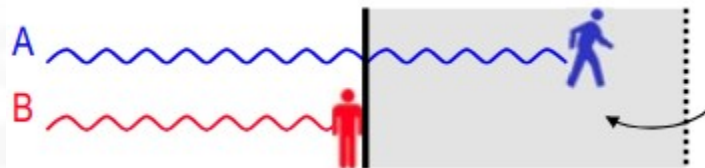


Região Crítica (mesmo código/rotina compartilhado entre processos)



Região Crítica (códigos distintos, mas acessando as mesmas variáveis/dados compartilhados)

# Exclusão Mútua



Região Crítica (mesmo código/rotina compartilhado entre processos)

```

void echo()
{
    char chin, chout;
    do {
        chin = getchar();
        chout = chin;
        putchar(chout);
    }
    while (...);
}

```

A

```

void echo()
{
    char chin, chout;
    do {
        chin = getchar();
        chout = chin;
        putchar(chout);
    }
    while (...);
}

```

B

# Exclusão Mútua

```
do {  
    entry section  
    critical section  
    exit session  
    remainder section  
} while (TRUE);
```



# Concorrência

- Dificuldades:
  - Compartilhamento de recursos globais.
  - Gerência de alocação de recursos.
  - Localização de erros de programação (depuração de programas).
- Ação necessária:
  - Proteger os dados compartilhados (variáveis, arquivos e outros recursos globais).
  - Promover o acesso ordenado (controle de acesso) aos recursos compartilhados ⇒ ***sincronização de processos.***

# Abordagens para Exclusão Mútua

- Requisitos para uma boa solução:
  - A apenas um processo é permitido estar dentro de sua R.C. num dado instante.
  - Nenhum processo que executa fora de sua região crítica deve bloquear outro processo (ex: processo pára fora da sua R.C.).
  - Nenhuma suposição pode ser feita sobre as velocidades relativas dos processos ou sobre o número de CPUs no sistema.
  - Nenhum processo pode ter que esperar eternamente para entrar em sua R.C. ou lá ficar eternamente.

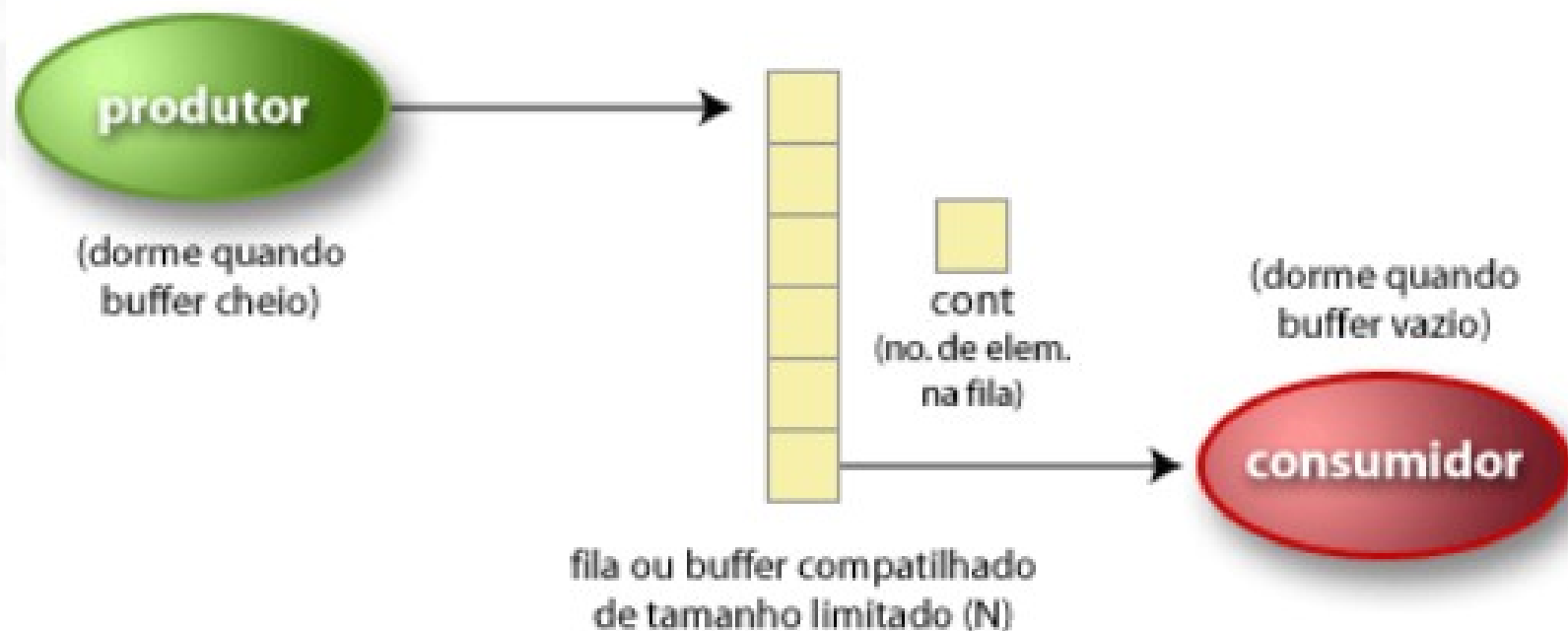
## Tipos de Soluções

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Decker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

## Inibição de Interrupções

- Usa um par de instruções do tipo DI / EI.
  - DI = *disable interrupt*      EI = *enable interrupt*
- O processo desativa todas as interrupções imediatamente antes de entrar na sua R.C., reativando-as imediatamente depois de sair dela.
- Com as interrupções desativadas, nenhum processo que está na sua R.C. pode ser interrompido, o que garante o acesso exclusivo aos dados compartilhados.

# Exemplo - Problema do produtor-consumidor



## Exemplo - Problema do produtor-consumidor

- variável  $N$  indica quantos itens ainda podem ser colocados no *buffer*.

### *Produtor*

**DI**

LDA N

DCR A

STA N

**EI**

### *Consumidor*

**DI**

LDA N

INC A

STA N

**EI**

## Problemas da Solução DI/EI

- É desaconselhável dar aos processos de usuário o poder de desabilitar interrupções.
- Não funciona com vários processadores.
  - Houve o conceito de “Global Interrupt Disabling”, mas o mesmo foi abandonado (baixa eficiência)
    - <https://www.halolinux.us/kernel-reference/global-interrupt-disabling.html>
- Inibir interrupções por um longo período de tempo pode ter consequências danosas. Por exemplo, perde-se a sincronização com os dispositivos periféricos.
  - OBS: inibir interrupções pelo tempo de algumas poucas instruções pode ser conveniente para o *kernel* (p.ex., para atualizar uma estrutura de controle).

## Tipos de Soluções

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Decker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores



## Soluções com *Busy Wait*

- *Busy wait* = espera ativa ou espera ocupada.
- Basicamente o que essas soluções fazem é:
  - Quando um processo quer entrar na sua R.C. ele verifica se a entrada é permitida. Se não for, ele espera em um laço (improdutivo) até que o acesso seja liberado.
    - Ex: **while (vez == OUTRO) do {nothing};**
  - Conseqüência: desperdício de tempo de CPU.
- Problema da **inversão de prioridade**:
  - Processo *LowPriority* está na sua R.C. e é interrompido. Processo *HighPriority* é selecionado mas entra em espera ativa. Nesta situação, o processo *LowPriority* nunca vai ter a chance de sair da sua R.C.

## 1a. Tentativa - Variável de Bloqueio

- Variável de bloqueio, compartilhada, indica se a R.C. está ou não em uso.
  - $turn = 0 \Rightarrow$  R.C. livre                       $turn = 1 \Rightarrow$  R.C. em uso
- Tentativa para  $n$  processos:

```
var turn: 0..1  
turn := 0
```

```
Process  $P_i$ :
```

```
...  
while turn = 1 do {nothing};  
turn := 1;  
< critical section >  
turn := 0;
```

```
...
```

## Problemas da 1a. Tentativa

- A proposta não é correta pois os processos podem concluir “simultaneamente” que a R.C. está livre, isto é, os dois processos podem testar o valor de *turn* antes que essa variável seja feita igual a *true* por um deles.

## Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

## 2a. Tentativa – Alternância Estrita

- Variável global indica de quem é a vez na hora de entrar na R.C.
- Tentativa para 2 processos:

```
var turn: 0..1;
```

```
P0:
```

```
...
```

```
while turn ≠ 0 do {nothing};
```

```
< critical section >
```

```
turn := 1;
```

```
...
```

```
P1:
```

```
...
```

```
while turn ≠ 1 do {nothing};
```

```
< critical section >
```

```
turn := 0;
```

```
...
```

## Problemas da 2a. Tentativa

- O algoritmo garante a exclusão mútua, mas obriga a alternância na execução das R.C.
- Não é possível a um mesmo processo entrar duas vezes consecutivamente na sua R.C.
  - Logo, a “velocidade” de entrada na R.C. é ditada pelo processo mais lento.
- Se um processo falhar ou terminar, o outro não poderá mais entrar na sua R.C., ficando bloqueado permanentemente.

## 3a. Tentativa

- O problema da tentativa anterior é que ela guarda a *identificação* do processo que pode entrar na R.C.
  - Entretanto, o que se precisa, de fato, é de informação de *estado* dos processos (i.e., se eles *querem* entrar na R.C.)
- Cada processo deve então ter a sua própria “chave de intenção”. Assim, se falhar, ainda será possível a um outro entrar na sua R.C.
- A solução se baseia no uso de uma variável **array** para indicar a intenção de entrada na R.C.

## 3a. Tentativa

- Antes de entrar na sua R.C, o processo examina a variável de tipo **array**. Se ninguém mais tiver manifestado interesse, o processo indica a sua intenção de ingresso ligando o bit correspondente na variável de tipo **array** e prossegue em direção a sua R.C.

```
var flag: array[0..1] of boolean;  
flag[0] := false; flag[1] := false;
```

Process P0:

```
...  
while flag[1] do {nothing};  
flag[0] := true;  
< critical section >  
flag[0] := false;  
...
```

Process P1:

```
...  
while flag[0] do {nothing};  
flag[1] := true;  
< critical section >  
flag[1] := false;  
...
```



## Problemas da 3a. Tentativa

- Agora, se um processo falha fora da sua R.C. não haverá nenhum problema, nenhum processo ficará eternamente bloqueado devido a isso. Entretanto, se o processo falhar dentro da R.C., o problema ocorre.
- Não assegura exclusão mútua, pois cada processo pode chegar à conclusão de que o outro não quer entrar e, assim, entrarem simultaneamente nas R.C.
  - Isso acontece porque existe a possibilidade de cada processo testar se o outro não quer entrar (comando *while*) antes de um deles marcar a sua intenção de entrar.

## 4a. Tentativa

- A ideia agora é que cada processo marque a sua intenção de entrar *antes* de testar a intenção do outro, o que elimina o problema anterior.
- É o mesmo algoritmo anterior, porém com uma troca de linha.

Process P0:

```
...  
flag[0] := true;  
while flag[1] do  
{nothing};  
< critical section >  
flag[0] := false;  
...
```

Process P1:

```
...  
flag[1] := true;  
while flag[0] do  
{nothing};  
< critical section >  
flag[1] := false;  
...
```

## Problemas da 4a. Tentativa

- Garante a exclusão mútua mas se um processo falha dentro da sua R.C. (ou mesmo após *setar* o seu *flag*) o outro processo ficará eternamente bloqueado.
- Uma falha fora da R.C. não ocasiona nenhum problema para os outros processos.
- Problemão:
  - Todos os processos ligam os seus *flags* para *true* (marcando o seu desejo de entrar na sua R.C.). Nesta situação todos os processos ficarão presos no *while* em um *loop* eterno (situação de *deadlock*).

## 5a. Tentativa

- Na tentativa anterior o processo assinalava a sua intenção de entrar na R.C. sem saber da intenção do outro, não havendo oportunidade dele mudar de ideia depois (i.e., mudar o seu estado para “*false*”).
- A 5a. tentativa corrige este problema:
  - Após testar no *loop*, se o outro processo também quer entrar na sua R.C, em caso afirmativo, o processo com a posse da CPU declina da sua intenção, dando a vez ao parceiro.

## 5a. Tentativa (cont.)

Process P0:

```
...  
flag[0] := true;  
while flag[1] do  
  begin  
    flag[0] := false;  
    <delay for a short time>  
    flag[0] := true  
  end;  
< critical section >  
flag[0] := false;  
...
```

Process P1:

```
...  
flag[1] := true;  
while flag[0] do  
  begin  
    flag[1] := false;  
    <delay for a short time>  
    flag[1] := true  
  end;  
< critical section >  
flag[1] := false;  
...
```

## 5a. Tentativa (cont.)

- Esta solução é quase correta. Entretanto, existe um pequeno problema: a possibilidade dos processos ficarem cedendo a vez um para o outro “indefinidamente” (problema da “mútua cortesia”)
  - Livelock
- Na verdade, essa é uma situação muito difícil de se sustentar durante um longo tempo na prática, devido às velocidades relativas dos processos. Entretanto, ela é uma possibilidade teórica, o que invalida a proposta como solução geral do problema.

## 5a. Tentativa – Exemplo

$P_0$  seta *flag[0]* para *true*.

$P_1$  seta *flag[1]* para *true*.

$P_0$  testa *flag[1]*.

$P_1$  testa *flag[0]*.

$P_0$  seta *flag[0]* para *false*.

$P_1$  seta *flag[1]* para *false*.

$P_0$  seta *flag[0]* para *true*.

$P_1$  seta *flag[1]* para *true*.

## Solução de Dekker

- Trata-se da primeira solução correta para o problema da exclusão mútua de dois processos (proposta na década de 60).
- O algoritmo combina as ideias de variável de bloqueio e *array* de intenção.
- É similar ao algoritmo anterior mas usa uma variável adicional (*vez/turn*) para realizar o desempate, no caso dos dois processos entrarem no *loop* de mútua cortesia.



# Algoritmo de Dekker

```
var flag: array[0..1] of boolean;  
    turn: 0..1; //who has the priority
```

```
flag[0] := false  
flag[1] := false  
turn := 0 // or 1
```

```
Process p0:  
    flag[0] := true  
    while flag[1] {  
        if turn ≠ 0 {  
            flag[0] := false  
            while turn ≠ 0 {}  
            flag[0] := true  
        }  
    }  
  
    // critical section  
    ...  
    // end of critical section  
    turn := 1  
    flag[0] := false
```

```
Process p1:  
    flag[1] := true  
    while flag[0] {  
        if turn ≠ 1 {  
            flag[1] := false  
            while turn ≠ 1 {}  
            flag[1] := true  
        }  
    }  
  
    // critical section  
    ...  
    // end of section  
    turn := 0  
    flag[1] := false
```

## Algoritmo de Dekker (cont.)

- Quando  $P0$  quer entrar na sua R.C. ele coloca seu *flag* em *true*. Ele então vai checar o *flag* de  $P1$ .
- Se o *flag* de  $P1$  for *false*, então  $P0$  pode entrar imediatamente na sua R.C.; do contrário, ele consulta a variável *turn*.
- Se  $turn = 0$  então  $P0$  sabe que é a sua vez de insistir e, deste modo, fica em *busy wait* testando o estado de  $P1$ .
- Em certo ponto,  $P1$  notará que é a sua vez de declinar. Isso permite ao processo  $P0$  prosseguir.
- Após  $P0$  usar a sua R.C. ele coloca o seu *flag* em *false* para liberá-la, e faz  $turn = 1$  para transferir o direito para  $P1$ .

## Algoritmo de Dekker (cont.)

- Algoritmo de Dekker resolve o problema da exclusão mútua
- 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?
  - Poderíamos nos dar 'ao luxo' de consumir ciclos de CPU,
  - Situação rara na prática (em geral, há mais processos do que CPUs)
  - 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.
- Devemos fazer uma modificação significativa do programa se quisermos estender a solução de 2 para N processos:
  - `flag[]` com N posições; variável `turn` passa a assumir valores de 1..N; alteração das condições de teste em todos os processos

## Solução de Peterson

- Proposto em 1981, é uma solução simples e elegante para o problema da exclusão mútua, sendo facilmente generalizado para o caso de  $n$  processos.
- O truque do algoritmo consiste no seguinte:
  - Ao marcar a sua intenção de entrar, o processo já indica (para o caso de empate) que a vez é do outro.
- Mais simples de ser verificado

# Algoritmo de Peterson

```
flag[0] := false
flag[1] := false
turn    := 0
```

Process P0:

```
    flag[0] := true
    turn := 1
    while ( flag[1] && turn == 1 ){
        // do nothing
    }
    // critical section
    ...
    // end of critical section
    flag[0] := false
```

Process P1:

```
    flag[1] := true
    turn := 0
    while ( flag[0] && turn == 0 ){
        // do nothing
    }
    // critical section
    ...
    // end of critical section
    flag[1] := false
```

## Solução de Peterson (cont.)

- Exclusão mútua é atingida.
  - Uma vez que  $P0$  tenha feito  $flag[0] = true$ ,  $P1$  não pode entrar na sua R.C.
  - Se  $P1$  já estiver na sua R.C., então  $flag[1] = true$  e  $P0$  está impedido de entrar.
- Bloqueio mútuo (deadlock) é evitado.
  - Supondo  $P0$  bloqueado no seu *while*, isso significa que  $flag[1] = true$  e que  $turn = 1$
  - se  $flag[1] = true$  e que  $turn = 1$ , então  $P1$  por sua vez entrará na sua seção crítica
  - Assim,  $P0$  só pode entrar quando **ou**  $flag[1]$  tornar-se *false* **ou**  $turn$  passar a ser 0.

## Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup, Semáforos, Monitores

A alteração do valor p/ “trancado” APÓS o teste permite que dois processos executem a R.C. ao mesmo tempo! O TESTE e a ALTERAÇÃO necessitam ser feitos de forma **indivisível...**

## A Instrução TSL (1)

- TSL = “Test and Set Lock”
- Solução de hardware para o problema da exclusão mútua em ambiente com vários processadores.
  - O processador que executa a TSL bloqueia o barramento de memória, impedindo que outras CPUs acessem a MP até que a instrução tenha terminado.
- A instrução TSL faz o seguinte:
  - Lê o conteúdo de um endereço de memória (variável compartilhada “*lock*”, usada para proteger a R.C.) para um registrador e armazena um valor diferente de zero (normalmente 1) nesse endereço.



## A Instrução TSL (2)

- Se  $lock = 0 \Rightarrow$  R.C. livre;  
Se  $lock = 1 \Rightarrow$  R.C. ocupada.  
(*Lock* é iniciada com o valor 0).
- A instrução TSL é executada de forma atômica.
  - As operações de leitura e armazenamento da variável *lock* são garantidamente indivisíveis, sem interrupção.
  - Nenhuma outra CPU pode acessar *lock* enquanto a instrução não tiver terminado.

## A Instrução TSL (3)

- Em ling. de alto nível, seria o mesmo que fazer o seguinte de forma atômica:

```
boolean testset (int lock) {  
    if (lock == 0) {  
        lock = 1;  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

## A Instrução TSL (4)

```
enter_region:
    tsl register, flag      | copia flag p/
                            | registrador e faz flag = 1
    cmp register, #0        | o flag é zero?
    jnz enter_region       | se não, lock e setado; loop
    ret                    | retorna, entrou na R.C.

leave_region:
    mov flag, #0           | guarda um 0 em flag
    ret                    | retorna a quem chamou
```

## A Instrução TSL (5)

- Vantagens da TSL:
  - Simplicidade de uso (embora sua implementação em hardware não seja trivial).
  - Não dá aos processos de usuário o poder de desabilitar interrupções.
  - Presente em quase todos os processadores atuais.
  - Funciona em máquinas com vários processadores.
- Desvantagens:
  - Espera ocupada (*busy wait*).
  - Possibilidade de postergação infinita (starvation)
    - “processo azarado” sempre pega a variável *lock* com o valor 1

## Referências Adicionais (Extra)

“Global Interrupt Disabling”

<https://www.halolinux.us/kernel-reference/global-interrupt-disabling.html>

Buhr, Peter A., David Dice, and Wim H. Hesselink. "**Dekker's mutual exclusion algorithm made RW-safe.**" *Concurrency and Computation: Practice and Experience* 28.1 (2016): 144-165.

<https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.3659>

Buhr, P. A., Dice, D., & Hesselink, W. H. (2015). High-performance N-thread software solutions for mutual exclusion. *Concurrency and Computation: Practice and Experience*, 27(3), 651-701.

<https://onlinelibrary.wiley.com/doi/full/10.1002/cpe.3263>