



Laboratório de Pesquisa em Redes e Multimídia

Sistemas Operacionais

Sincronização de processos (1): mecanismos de busy wait



Universidade Federal do Espírito Santo
Departamento de Informática

Motivação ⁽¹⁾

Considere 2 processos que compartilham memória:

program 0

```
{  
  *  
  *  
  counter++  
  *  
}
```

Shared variable

```
int counter=5;
```

program 1

```
{  
  *  
  *  
  counter--  
  *  
}
```

Motivação (2)

program 0

```
{  
  *  
  *  
  counter++  
  *  
}
```

Shared variable

```
int counter=5;
```

program 1

```
{  
  *  
  *  
  counter--  
  *  
}
```

- What is the value of counter?
 - expected to be 5
 - but could also be 4 and 6

Motivação ⁽³⁾

program 0

```
{  
  *  
  *  
  counter++  
  *  
}
```

Shared variable

```
int counter=5;
```

program 1

```
{  
  *  
  *  
  counter--  
  *  
}
```

context switch

```
R1 ← counter  
R1 ← R1 + 1  
counter ← R1  
-----  
R2 ← counter  
R2 ← R2 - 1  
counter ← R2
```

counter = 5

A operação "counter++" em C é traduzida para essas 3 instruções de máquina

A operação "counter--" em C é traduzida para essas 3 instruções de máquina

Motivação ⁽³⁾

program 0

```
{  
  *  
  *  
  counter++  
  *  
}
```

Shared variable
int counter=5;

program 1

```
{  
  *  
  *  
  counter--  
  *  
}
```

context switch

```
R1 ← counter  
R1 ← R1 + 1  
counter ← R1  
-----  
R2 ← counter  
R2 ← R2 - 1  
counter ← R2
```

counter = 5

Neste cenário, P0 executa as três instruções indicadas e é preemptado. Na sequência, P1 é escalonado e executa as três instruções indicadas. Até aqui tudo certo... counter termina com o valor 5!

Motivação ⁽⁴⁾

program 0

```
{
 *
 *
 counter++
 *
}
```

Shared variable

```
int counter=5;
```

program 1

```
{
 *
 *
 counter--
 *
}
```

context
switch

```
R1 ← counter
R1 ← R1 + 1
counter ← R1
R2 ← counter
R2 ← R2 - 1
counter ← R2
```

counter = 5

```
R1 ← counter
R2 ← counter
R2 ← R2 - 1
counter ← R2
R1 ← R1 + 1
counter ← R1
```

counter = 6

Mas agora considere que por causa do escalonamento, a sequência de execução seja esta:

- P0 roda uma instrução e é preemptado
- P1 roda as três instruções e é preemptado
- P0 roda mais 2 instruções

... counter igual a 6 !!!!!

Motivação ⁽⁵⁾

program 0

```
{  
  *  
  *  
  counter++  
  *  
}
```

Shared variable

```
int counter=5;
```

program 1

```
{  
  *  
  *  
  counter--  
  *  
}
```

context
switch

```
R1 ← counter  
R1 ← R1 + 1  
counter ← R1  
R2 ← counter  
R2 ← R2 - 1  
counter ← R2
```

counter = 5

```
R1 ← counter  
R2 ← counter  
R2 ← R2 - 1  
counter ← R2  
R1 ← R1 + 1  
counter ← R1
```

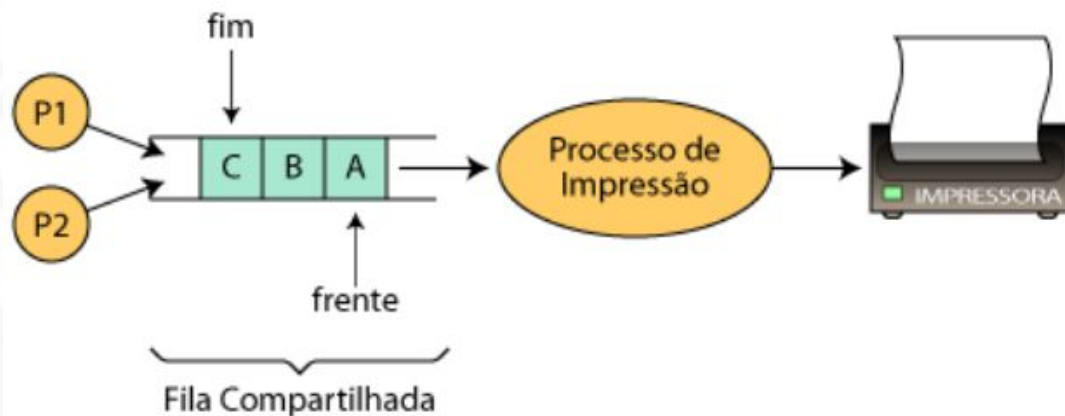
counter = 6

```
R2 ← counter  
R2 ← counter  
R2 ← R2 + 1  
counter ← R2  
R2 ← R2 - 1  
counter ← R2
```

counter = 4

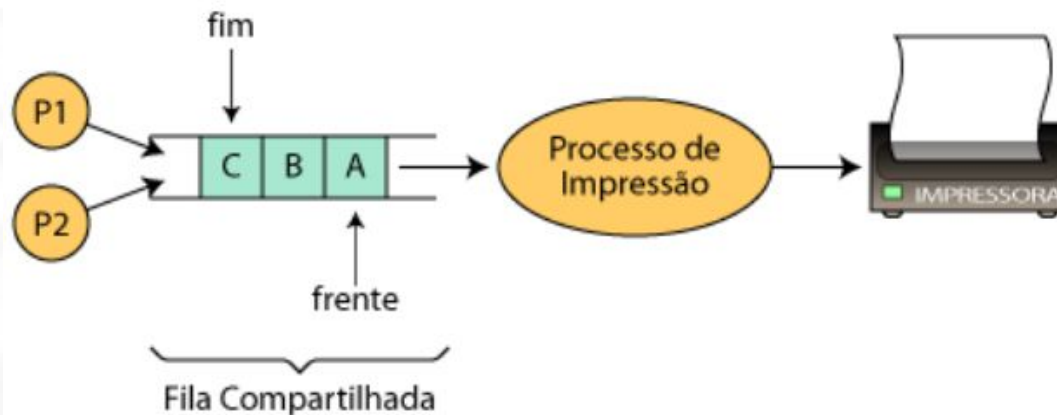
... ou 4 !!!

Mais Exemplos (1)



- 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

Mais Exemplos (2)

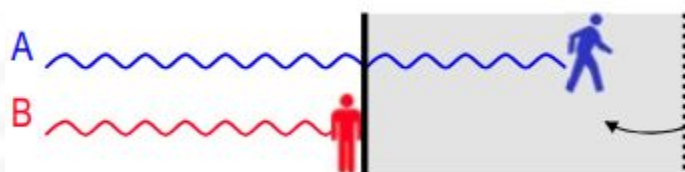


1. `fim++` (incrementa o indicador do fim da fila)
 2. coloca documento na posição do novo fim da fila
- dois processos resolvem simultaneamente imprimir um documento
 - o primeiro processo foi interrompido (por ter acabado o seu quantum) entre os comandos 1 e 2
 - o segundo processo insere seu documento 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

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

Região Crítica (1)



Região Crítica (mesmo código/rotina compartilhado entre diferentes processos)

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

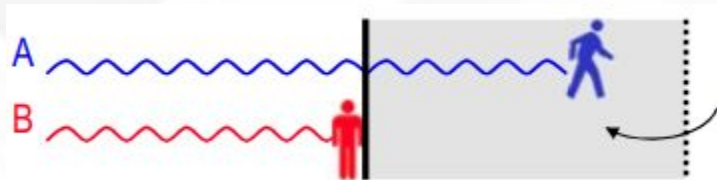
A

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

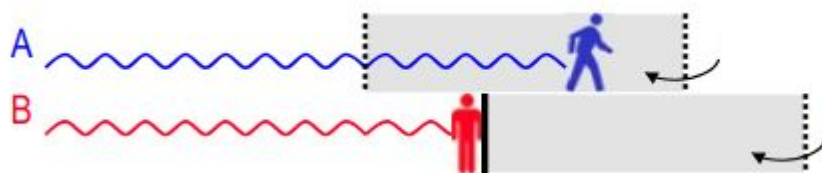
B

Região Crítica ⁽¹⁾

- Podem ocorrer **Condições de corrida** em duas situações diferentes:



Região Crítica (mesmo código/rotina compartilhado entre processos). Se mais de um processo executa essa mesma rotina ao mesmo tempo, temos condição de corrida...



Também podemos ter regiões críticas em códigos distintos, mas que acessam as mesmas variáveis/dados compartilhados...

Exclusão Mútua ... é a solução para evitar condições de corrida!!

```
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 para 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.

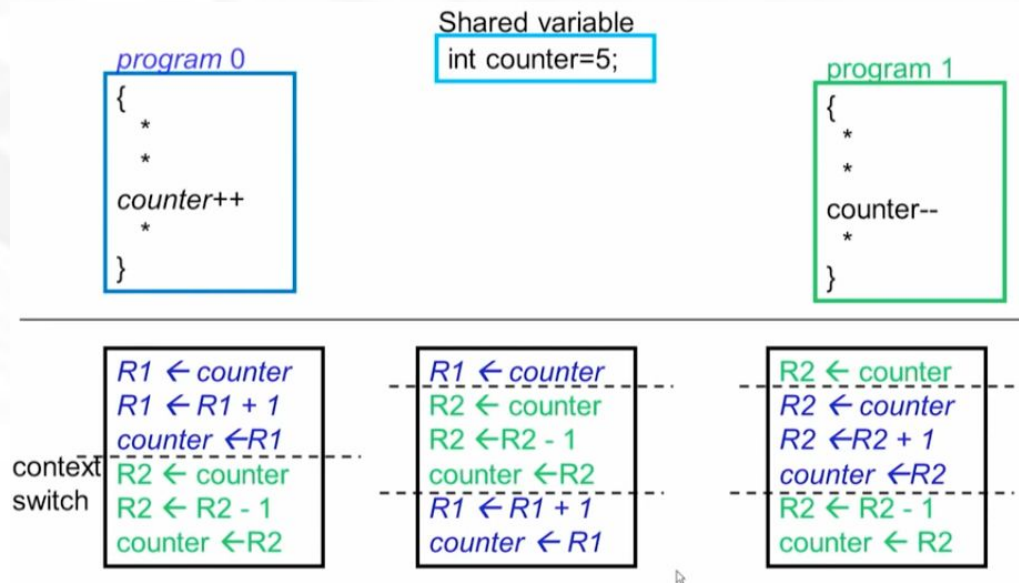
Tipos de Soluções

- Soluções de Hardware
 - Inibição de interrupções
 - Instrução TSL
- 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 ⁽¹⁾

- 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.

Inibição de Interrupções (2)



shared int counter;

Code for p₁

```
disableInterrupts();
counter++;
enableInterrupts();
```

Code for p₂

```
disableInterrupts();
counter--;
enableInterrupts();
```

Problemas da Solução DI/EI

- É desaconselhável dar aos processos de usuário o poder de desabilitar interrupções.
- **Não funciona com vários processadores.**
- Perda de sincronização com dispositivos periféricos
 - 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).

A Instrução TSL (Test and Set Lock) ⁽¹⁾

- Solução de hardware para o problema da exclusão mútua em ambiente com vários processadores.
- Ideia básica: uso de uma variável de bloqueio ("lock"), que é manipulada por uma instrução da CPU.
 - Se lock = 0, a R.C. está livre
 - Se lock != 0, a R.C. está ocupada

Process 1

```
while(1){  
    while(lock != 0);  
    lock = 1; // lock  
    critical section  
    lock = 0; // unlock  
    other code  
}
```

lock=0

Process 2

```
while(1){  
    while(lock != 0);  
    lock = 1; // lock  
    critical section  
    lock = 0; // unlock  
    other code  
}
```


A Instrução TSL (Test and Set Lock) (2)

Process 1

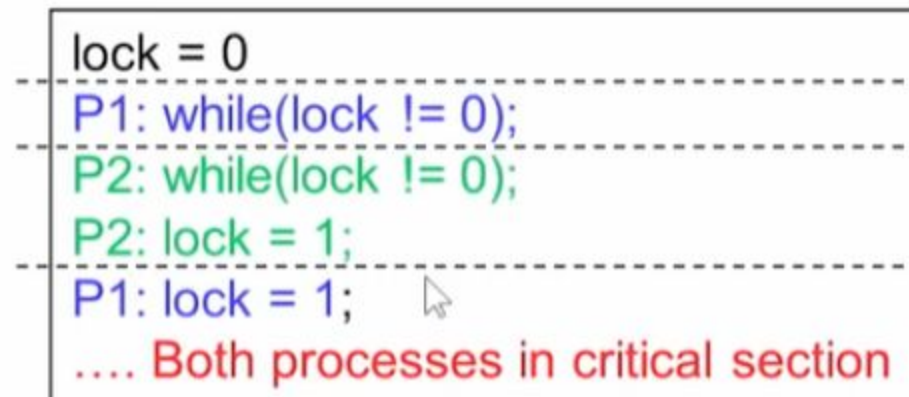
```
while(1){  
  while(lock != 0);  
  lock = 1; // lock  
  critical section  
  lock = 0; // unlock  
  other code  
}
```

lock=0

Process 2

```
while(1){  
  while(lock != 0);  
  lock = 1; // lock  
  critical section  
  lock = 0; // unlock  
  other code  
}
```

No



context switch

A Instrução TSL (Test and Set Lock) (3)

Process 1

```
while(1){  
    while(lock != 0);  
    lock= 1; // lock  
    critical section  
    lock = 0; // unlock  
    other code  
}
```

Make atomic

The diagram illustrates the implementation of a Test and Set Lock (TSL) instruction using a while loop. A blue box highlights the entire loop body. A dashed rectangle within the loop body encloses the lock acquisition and the critical section. A blue arrow points from the text 'Make atomic' to the dashed rectangle, indicating that the operations within it must be atomic. The code shows a process entering a loop, waiting for the lock to be free, acquiring it, executing the critical section, and then unlocking it before executing other code.

A Instrução TSL (Test and Set Lock) (4)

```
while(1){  
    while(test_and_set(&lock) == 1);  
    critical section  
    lock = 0; // unlock  
    other code  
}
```

Why does this work? If two CPUs execute `test_and_set` at the same time, the hardware ensures that one `test_and_set` does both its steps before the other one starts.

So the first invocation of `test_and_set` will read a 0 and set lock to 1 and return. The second `test_and_set` invocation will then see lock as 1, and will loop continuously until lock becomes 0

A Instrução TSL (Test and Set Lock) (5)

```
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 é 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 (Test and Set Lock) (6)

- 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:
 - Apresenta espera ocupada (*busy wait*), desperdiçando ciclos da CPU (mas pode ser uma boa solução se a espera não for longa).
 - Possibilidade de postergação infinita (starvation)
 - "Processo azarado" sempre pega a variável *lock* com o valor 1

Soluções "Spinlocks"

- Characteristic : **busy waiting**
 - Useful for short critical sections, where much CPU time is not wasted waiting
 - eg. To increment a counter, access an array element, etc.
 - Not useful, when the period of wait is unpredictable or will take a long time
 - eg. Not good to read page from disk.
 - Use mutex instead (...mutex)

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

- Não é uma solução do problema.
- 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.
 - O requisito de progresso não é atendido.
- 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 ⁽¹⁾

- 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 (2)

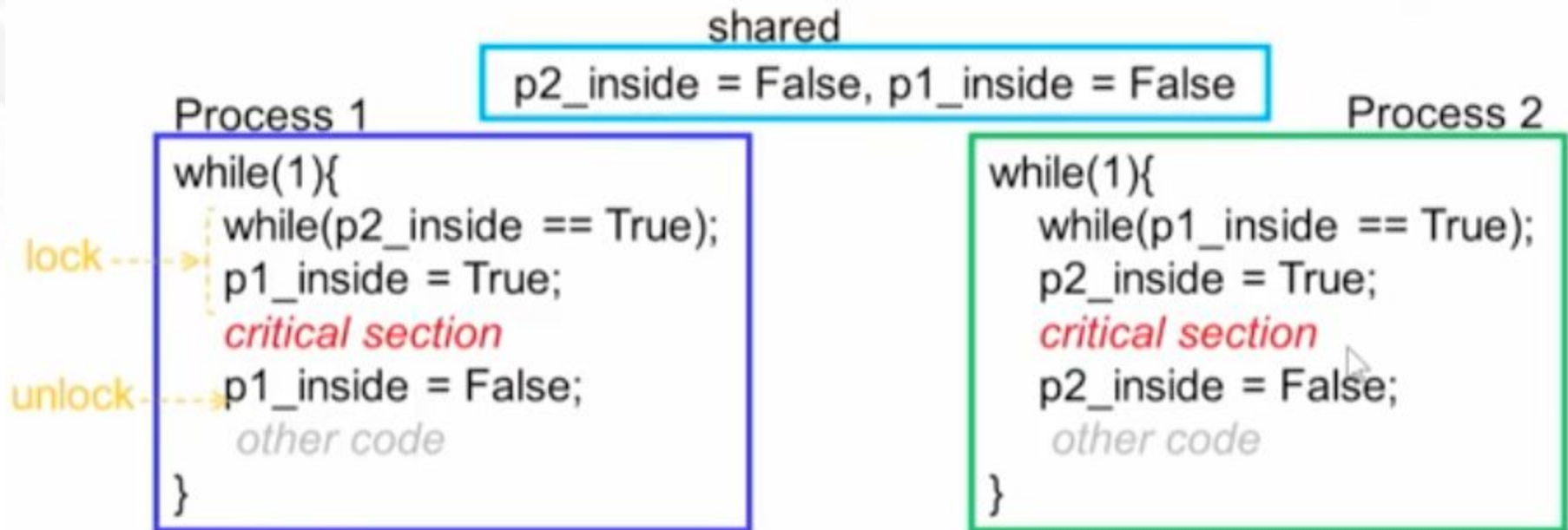
- Antes de entrar na sua R.C, o processo examina a variável de tipo **array**. Se ninguém mais tiver manifestado interesse, o processo indica a sua intenção de ingresso ligando o bit correspondente na variável de tipo **array** e prossegue em direção a sua R.C.

```
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;  
...
```

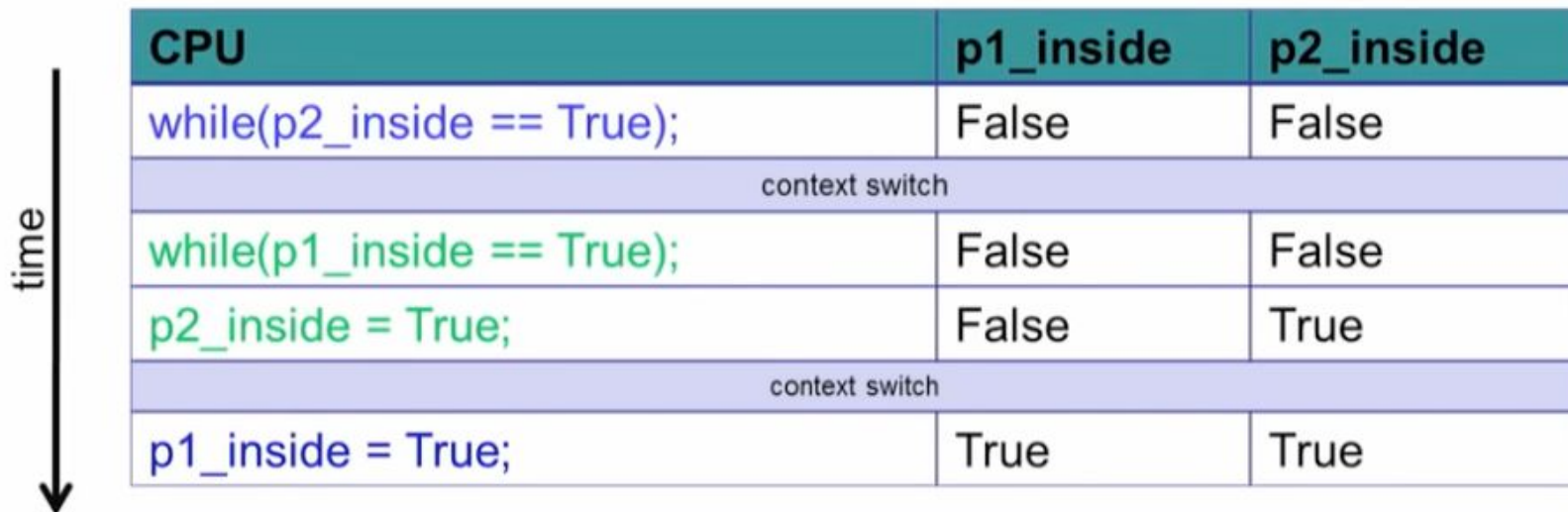
3a. Tentativa (3)



Problemas da 3a. Tentativa ⁽¹⁾

- 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.

Problemas da 3a. Tentativa (2)



CPU	p1_inside	p2_inside
<code>while(p2_inside == True);</code>	False	False
context switch		
<code>while(p1_inside == True);</code>	False	False
<code>p2_inside = True;</code>	False	True
context switch		
<code>p1_inside = True;</code>	True	True

Both p1 and p2 can enter into the critical section at the same time

```
while(1){  
    while(p2_inside == True);  
    p1_inside = True;  
    critical section  
    p1_inside = False;  
    other code  
}
```

```
while(1){  
    while(p1_inside == True);  
    p2_inside = True;  
    critical section  
    p2_inside = False;  
    other code  
}
```


4a. Tentativa ⁽¹⁾

- 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 ⁽¹⁾

- 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 (2)

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 ⁽³⁾

- 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 (4)

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*.

Algoritmo de Dekker ⁽¹⁾

- 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 (2)

```
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 ⁽³⁾

- 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 (4)

- 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 ⁽¹⁾

- Algoritmo proposto em 1981, é uma solução simples e elegante para o problema da exclusão mútua, sendo facilmente generalizada 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

Solução de Peterson (2)

```
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 (3)

- 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.

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>