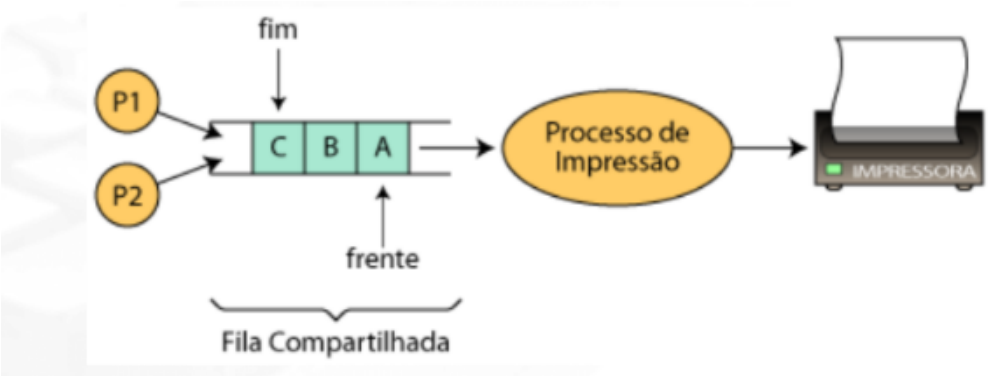
Condições de Corrida

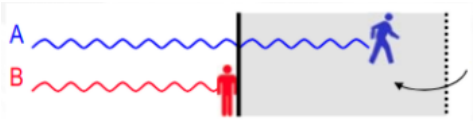
São situações em que dois ou mais processos acessam dados compartilhados e o resultado final depende da ordem em que os processos são executados, ditada pelo mecanismo de escalonamento do SO, ou seja, dependendo da ordem que os processos são executados pode haver ou não inconsistências.

Para evitar essas condições de corridas, é necessário que o programador impeça de alguma forma que dois ou mais processos acessem a região crítica.

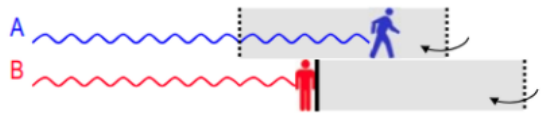


Região Crítica

É a parte do programa em que os dados compartilhados são acessados e onde ocorrem as condições de corrida.



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

A exclusão mútua tem como objetivo proibir que mais de um processo entre em sua Região Crítica, através da exclusão mútua, essas condições de corrida são evitadas, pois ela garante que somente um processo estará usando os dados compartilhados num dado momento., ou seja, se um processo entrar na região crítica.. os demais tem que esperar.

Tipos de Soluções

Soluções de Hardware (da CPU) - Inibição de Interrupções

A inibição de Interrupções se resume a um par de instruções do tipo DI = disable interrupt e EI = enable interrupt, através dela 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.

É desaconselhável dar aos processos de usuário o poder de desabilitar essas interrupções, pois um processo mal intencionado poderia desabilitar a interrupção e não habilitar de novo.. virando um tipo de “deus” na máquina e nem o SO iria poder entrar mais, além de não funcionar com vários processadores há a perda de sincronização com dispositivos periféricos.

Soluções de Hardware (da CPU) - A Instrução TSL (Test and Set Lock)

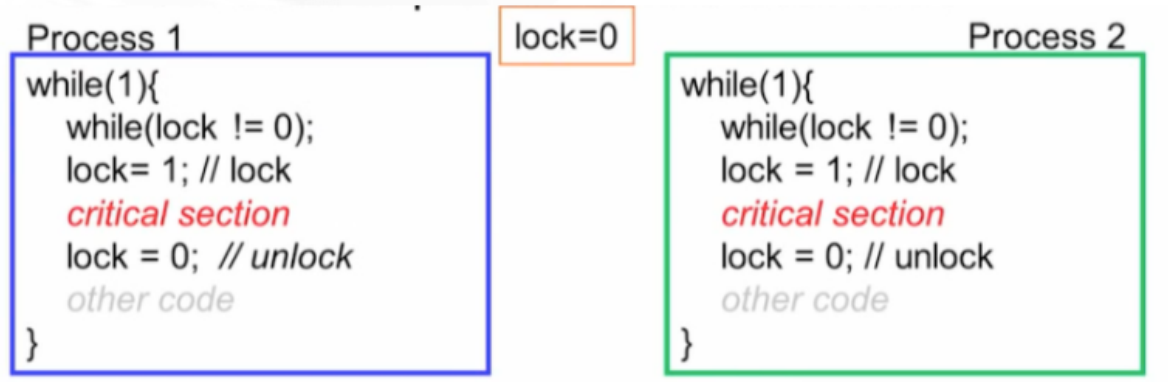
É um tipo de solução de hardware para o problema da exclusão mútua em ambiente com vários processadores.

Sua ideia básica se resume ao 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

Irá se comportar como se fosse um cadeado, se tiver aberto quer dizer que a região critica estaria live e qualquer processo poderia ocupar esse espaço, mas entes de entrar nessa R.C o processo fecharia o cadeado, permitindo que apenas fique na RC, o problema desse comportamento se dá quando um processo perde a posse da CPU entre o tempo de chegar a conclusão que a região critica esta livre e setar a variavel do testada no loop pra 1.



Embora sua implementação em hardware não seja trivial, há simplicidade em seu uso e além de não dá aos processos de usuário o poder de desabilitar interrupções essa solução funciona em máquinas com vários processadores, entretanto, apresenta espera ocupada (busy wait), desperdiçando ciclos da CPU com possibilidade de postergação infinita (starvation “Processo azarado” sempre pega a variável lock com o valor 1)

Soluções com Busy Wait

Basicamente, 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, tendo como consequência odesperdício de tempo de CPU. .

Ex: while (vez == OUTRO) do {nothing};

Entretanto, nessas soluções ocorrem o problema da inversão de prioridade, situações em que processos Low Priority estão interrompidos e processos High Priority são selecionados mas entra na espera ativa, os processo Low Priority nunca vai ter a chance de sair da sua R.C, para isso não acontecer, existem algumas opções:

Variável de Bloqueio:

Indica se a R.C. está ou não em uso.

turn = 0 ⇒ R.C. livre

turn = 1 ⇒ R.C. em uso

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

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

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.

Alternância Estrita:

Funciona com dois processos, uma variável global indica de quem é a vez na hora de entrar na R.C.

```
var turn: 0..1;

P0:                                  P1:
...                                  ...
while turn ≠ 0 do {nothing};         while turn ≠ 1 do {nothing};
< critical section >                 < critical section >
turn := 1;                           turn := 0;
...                                  ...
```

O algoritmo garante a exclusão mútua, mas obriga a alternância na execução das R.C, caso um processo venha falhar ou terminar, o outro não poderá mais entrar na sua R.C., ficando bloqueado permanentemente e não é possível a um mesmo processo entrar duas vezes consecutivamente.

Algoritmo de Dekker

Trata-se da primeira solução correta para o problema da exclusão mútua de dois processos, o algoritmo combina as ideias de variável de bloqueio e array de intenção, mas usa uma variável adicional (vez/turn) para realizar o desempate, no caso dos dois processos entrarem no loop. Esse tipo de solução só é aceitável se houver um número de CPUs igual (ou superior) ao número de processos que se devam executar no sistema e por isso é pouco usada.

Solução de Peterson

É uma solução simples para o problema da exclusão mútua, sendo facilmente generalizada para o caso de N processos, e sua ideia é basicamente marcar a sua intenção de entrar, e assim, o processo já indica (para o caso de empate) que a vez é do outro.

Nessa solução o programador não precisa utilizar nenhum tipo de recurso de máquina (hardware), tudo fica a nivel de software sendo executados em modo usuário e sua grande desvantagem é por usar a espera ocupada (Busy wait, ela fica presa em um loop até entrar na região crítica e se a máquina possui apenas um processador sendo compartilhado isso é muito ruim, pois realizando esse teste irá ocorrer um desperdício de CPU, logo ela é apenas interessante se haver uma cpu dedicada a cada processo.