COMUNICAÇÃO ENTRE PROCESSOS

DCE131 - Sistemas Operacionais

Atualizado em: 15 de maio de 2023

Iago Carvalho

Departamento de Ciência da Computação

COMUNICAÇÃO ENTRE PROCESSOS

Em diversas situações é desejado (ou necessário) que processos troquem mensagens entre si

De forma semelhante, também existe a necessidade de que *threads* realizem o compartilhamento de informações

Apesar de útil e necessário, esse compartilhamento de informações ocasiona alguns problemas de acesso e de segurança

- Tanto entre processos como entre *threads*

CONDIÇÃO DE DISPUTA

É o principal associado ao compartilhamento de informações

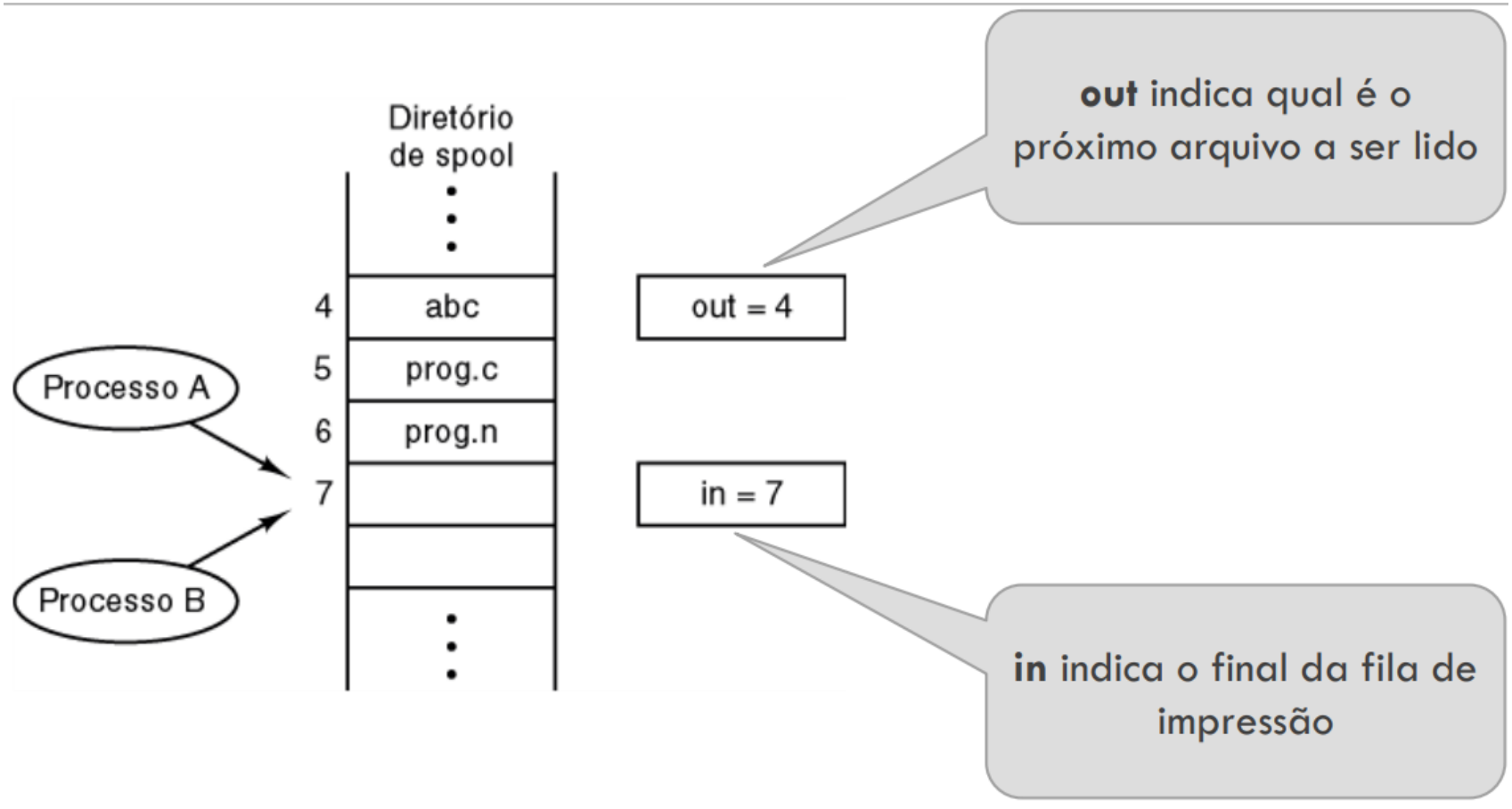
Dois ou mais processos e *threads* compartilham acesso ao mesmo recurso

- No geral, uma posição de memória
- Pode ser também um recurso compartilhado por rede

Vamos imaginar esta condição de disputa utilizando uma fila de impressão

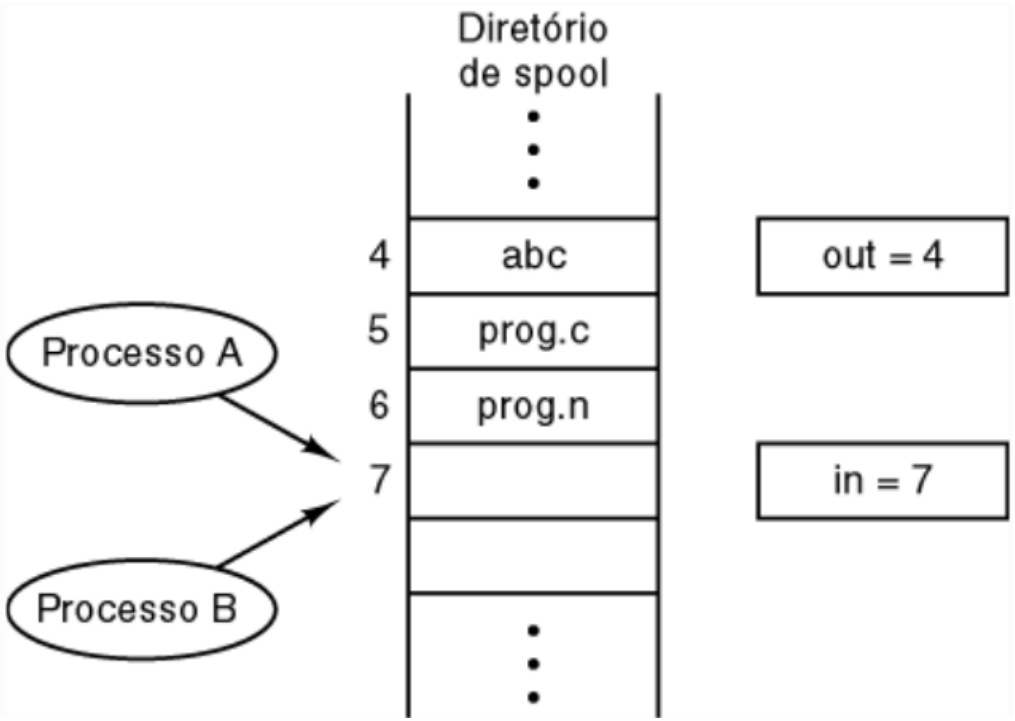
- Uma impressora possui um *buffer* de impressão
- Diferentes processos tem acesso a fila para indicar o que deve ser impresso
- Todos os processos podem requisitar impressões
  - Adicionar itens a fila

FILA DE IMPRESSÃO



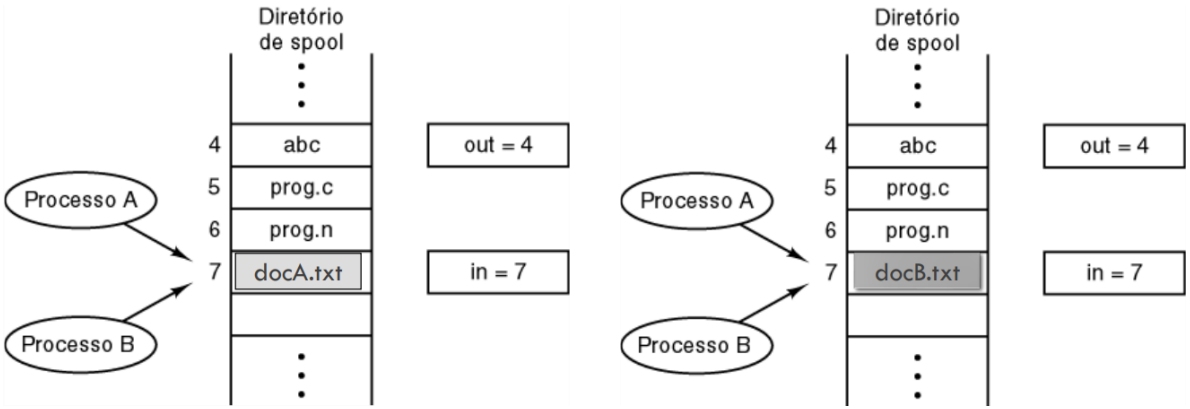
FILA DE IMPRESSÃO

Dois processos tentam, ao mesmo tempo, inserir um arquivo na fila de impressão



FILA DE IMPRESSÃO

1. Inicialmente, o processo *B* lê a variável *in* e captura a posição 7
2. Logo após, o processo *B* é preemptado e o processo *A* captura *in*
3. O processo *A* insere seu documento na posição 7
4. Processo *A* é preemptado do processador e o processo *B* volta a execução
5. Processo *B* insere seu documento na posição 7
6. O que acontece com o documento de *A*?



REGIÃO CRÍTICA

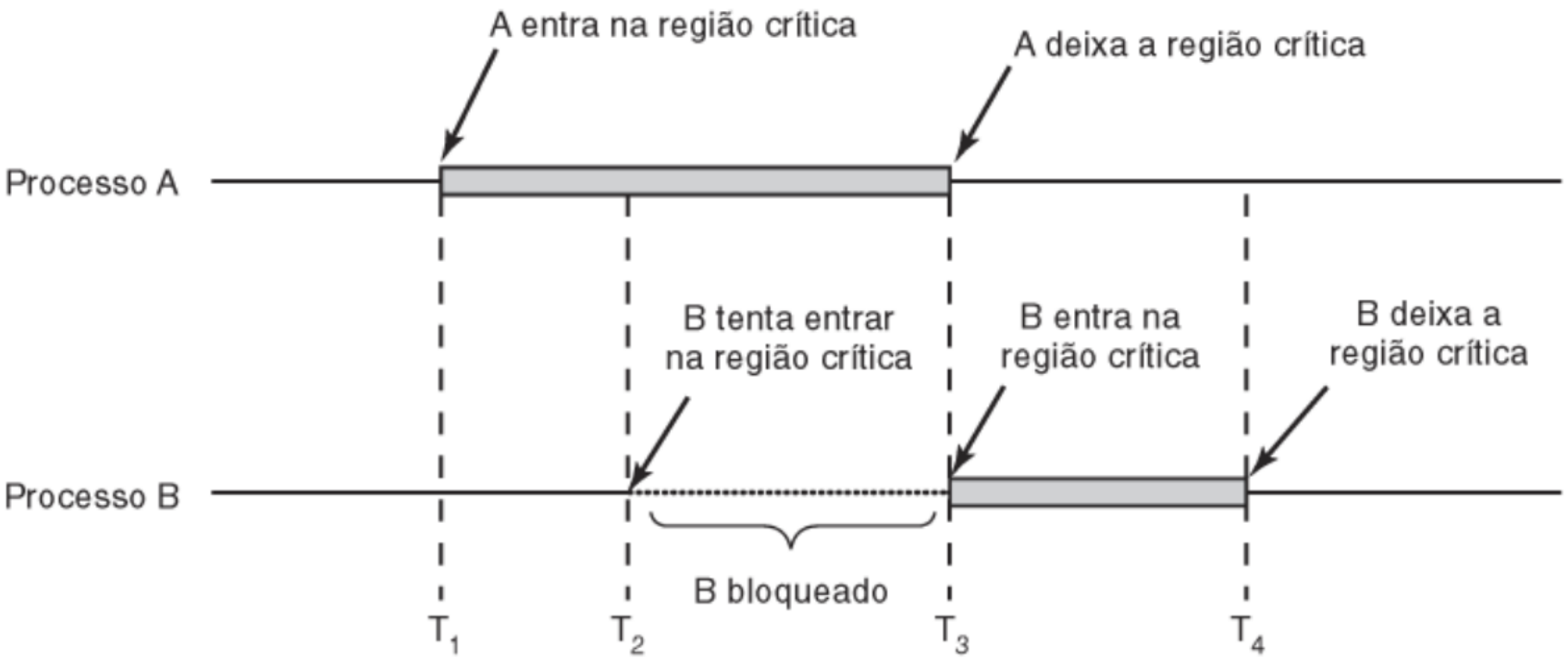
Dizemos que a fila de impressora é uma **região crítica**

Nestas regiões críticas, não é permitido a escrita simutânea

- Leitura é permitida

É necessário algum mecanismo por parte do Sistema Operacional para controlar o acesso a essa região crítica

EXCLUSÃO MÚTUA



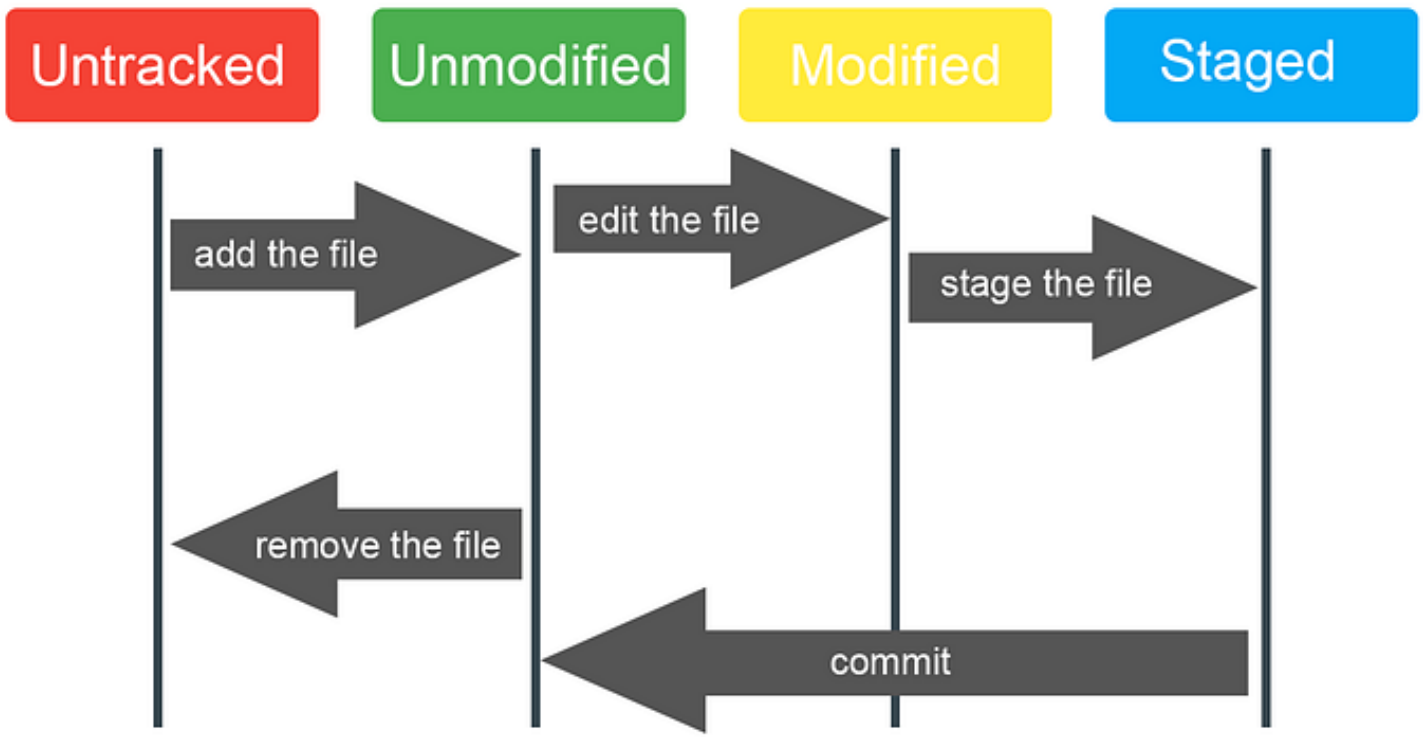
EXCLUSÃO MÚTUA

Um algoritmo para realizar a exclusão mútua deve garantir que

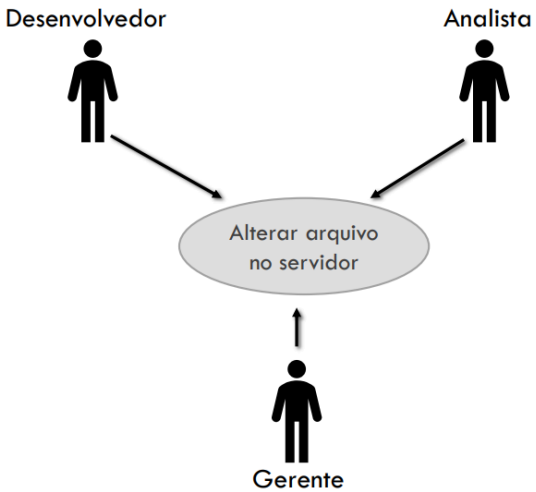
- Nunca dois processos podem estar simultaneamente na região crítica.
- Nada pode ser afirmado sobre a velocidade e/ou número de processadores.
- Nenhum processo executando fora da sua região crítica pode bloquear outros processos.
- Nenhum processo deve esperar eternamente para entrar em sua região crítica.

REGIÃO CRÍTICA EM UM CÓDIGO COMPARTILHADO

Onde vocês imaginam que está a sessão crítica em um código compartilhado?



REGIÃO CRÍTICA EM UM CÓDIGO COMPARTILHADO



1. Analista abre o arquivo *file.cpp*
2. Desenvolvedor também abre o arquivo *file.cpp*
3. Analista corrige um bug e salva o arquivo *file.cpp*
4. Desenvolvedor implementa uma nova funcionalidade
5. Desenvolvedor salva o arquivo *file.cpp*
6. Código continua com bug
7. Gerente fica puto porquê o cliente ta reclamando que o código tem bug

EXCLUSÃO MÚTUA UTILIZANDO ESPERA OCIOSA

Uma maneira de realizar a exclusão mútua é desabilitando as interrupções de sistema

- Necessário ser implementado em hardware

Antes de acessar uma região crítica, um processo desabilita o *clock* do sistema

- Isso interrompe o escalonador de processos

Ao fim do acesso, o processo habilita novamente o *clock*

EXCLUSÃO MÚTUA UTILIZANDO ESPERA OCIOSA

Vantagens: Implementa a exclusão mútua de forma perfeita

Desvantagens:

- Nenhum outro processo é executado
- Multiprogramação não existe mais
- Pipeline do processador é interrompido
- Se o processo trava, ele nunca mais sai do processador
  - Todo o sistema fica inutilizado
- Qualquer desenvolvedor pode fazer um processo que inutiliza todo o sistema

EXCLUSÃO MÚTUA UTILIZANDO ESPERA OCIOSA

Também existe uma opção (muito mais viável) via software

- **variáveis de bloqueio**

Utiliza-se uma variável compartilhada para indicar se uma região crítica está sendo utilizada ou não

- **lock = true**: região crítica está ocupada
- **lock = false**: região crítica está livre

Esta solução funciona?

EXCLUSÃO MÚTUA UTILIZANDO ESPERA OCIOSA

Ainda não... Vamos voltar para o exemplo da fila de impressão

1. O processo *A* lê a variável *in*
2. O processador preempta o processo *A* antes dele poder trocar o valor de *lock*
3. O processo *B* entra no processador e lê a variável *in*
4. Processo *B* verifica que *lock* == *false*, então ele faz alterações

ALGORITMO DE DE PETERSON

Esta é a primeira solução que realmente funciona

- Desenvolvido na década de 60
- Funciona, mas ainda não é perfeita

Sempre que um processo quer entrar na região crítica

- **enter_region(int pid)**

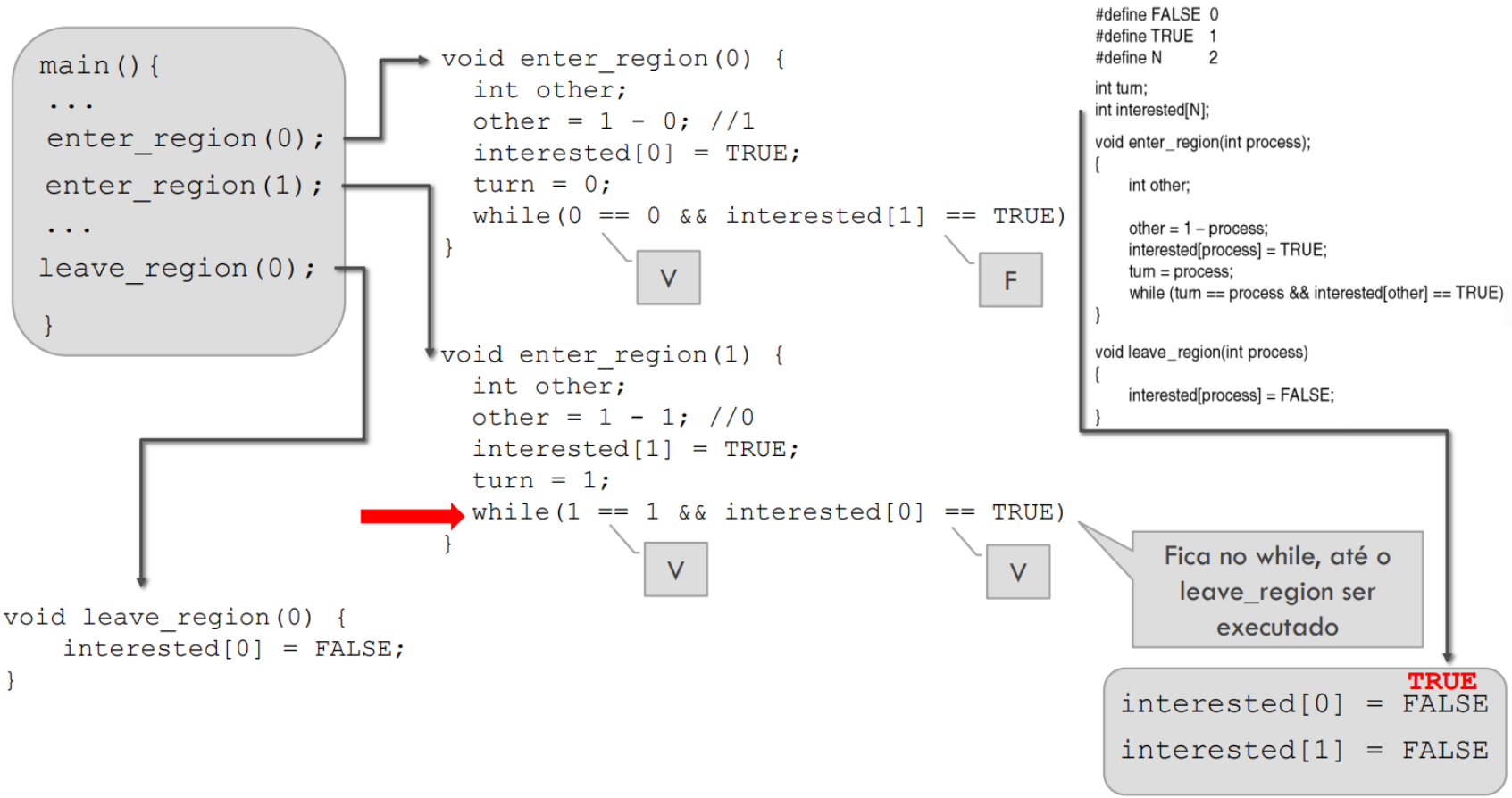
Sempre que um processo decide deixar a região crítica

- **leave_region(int pid)**

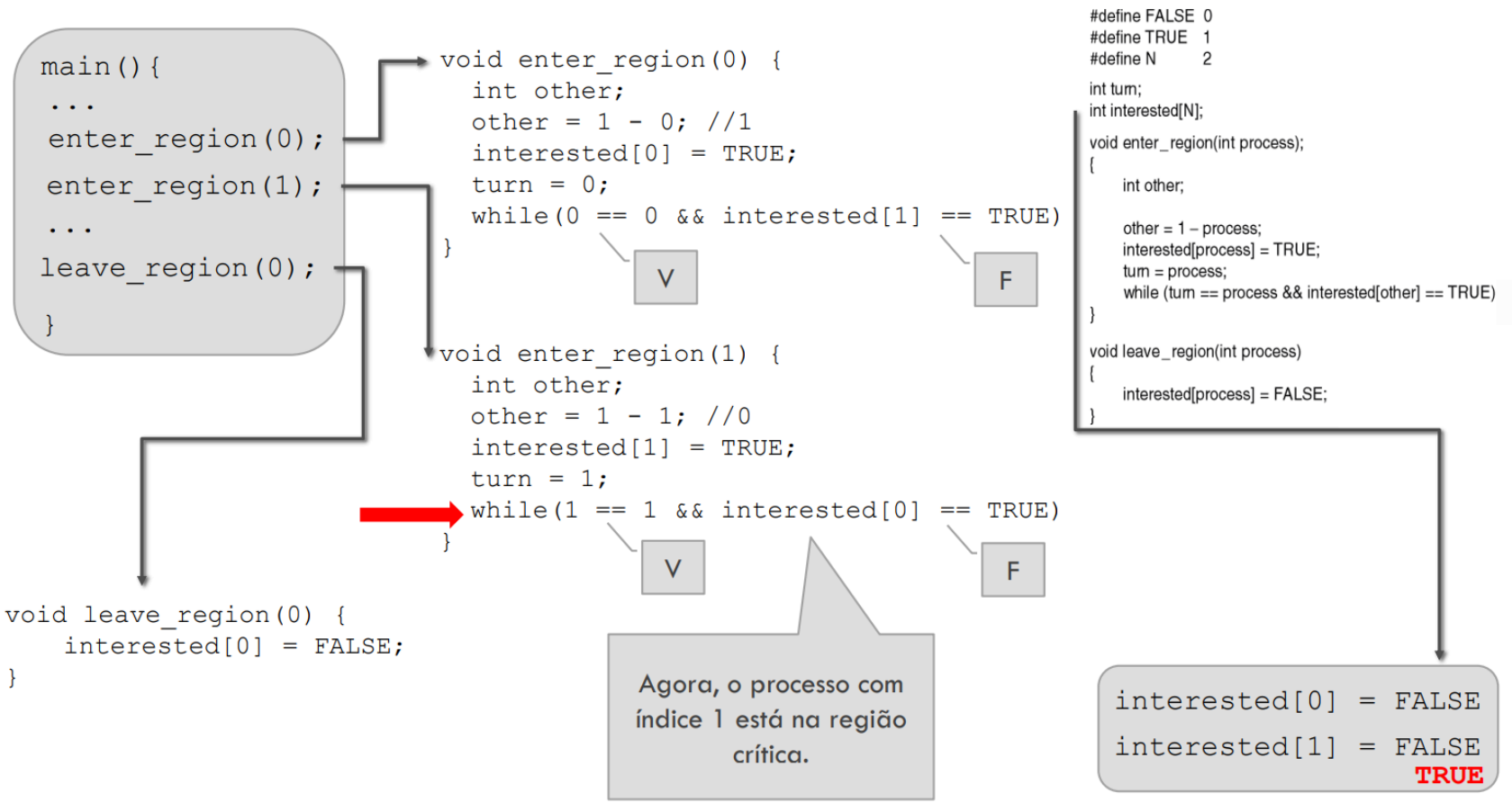
ALGORITMO DE PETERSON

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* número de processos */

int turn;                               /* de quem é a vez? */
int interested[N];                      /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process);         /* processo é 0 ou 1 */
{
      int other;                        /* número de outro processo */

      other = 1 – process;              /* o oposto do processo */
      interested[process] = TRUE;       /* mostra que você está interessado */
      turn = process;                   /* altera o valor de turn */
      while (turn == process && interested[other] == TRUE) //* comando nulo */;
}

void leave_region(int process)          /* processo: quem está saindo */
{
      interested[process] = FALSE;      /* indica a saída da região crítica */
}
```

ALGORITMO DE PETERSON



ALGORITMO DE PETERSON



E AGORA, TUDO CERTO?

ALGORITMO DE PETERSON

Ainda não...

Apesar da solução de Peterson resolver por completo o problema da região crítica, não deixando nenhuma possibilidade de acesso indevido, ela possui um defeito considerável

- Espera ociosa

PROBLEMA COM O ALGORITMO DE PETERSON

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* número de processos */

int turn;                           /* de quem é a vez? */
int interested[N];                  /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process);     /* processo é 0 ou 1 */
{
    int other;                      /* número de outro processo */

    other = 1 – process;            /* o oposto do processo */
    interested[process] = TRUE;     /* mostra que você está interessado */
    turn = process;                 /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE) //* comando nulo */;
}

void leave_region(int process)      /* processo: quem está saindo */
{
    interested[process] = FALSE;    /* indica a saída da região crítica */
}
```

PROBLEMA COM O ALGORITMO DE PETERSON

O processo que está aguardando ainda gasta tempo de processador

- Ele fica preso no *while*

E agora, como continuar?

- Semáforos (*sleep* e *weakup*)
- Monitores