Centro de Ciências Exatas e Tecnológicas Universidade Federal de Viçosa

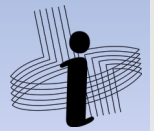
INF 310 – Programação Concorrente e Distribuída

Exclusão Mútua

Professor: Vitor Barbosa Souza
vitor.souza@ufv.br

Caracterização do Problema

- Programas concorrentes compartilham recursos
- Uso simultâneo de recursos compartilhados podem levar a erros na execução
  - Atualização + atualização
  - Leitura + atualização



UFV

Caracterização do Problema

- Ex.: Processos concorrentes compartilham 5 instâncias de um recurso

```
R : [5, 4, 3, 2, 1]; //vetor estático com id dos itens
T : 5;               //número de itens disponíveis
```

  - Requisição do recurso:

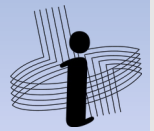
```
void requisita(&U) {
    T = T-1;
    U = R[T];
}
```

  - Liberação do recurso

```
void libera(&U) {
    R[T] = U;
    T = T+1;
}
```

  - Código de um processo

```
requisita(x);
// usa instância x
libera(x);
```



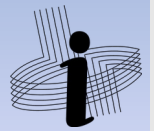
UFV

Caracterização do Problema

- Suponha que as 5 instâncias foram alocadas e depois liberadas as instâncias 3, 1 e 4, nesta ordem.
- Assim, os valores de R e T são:

```
R : [3, 1, 4, 2, 1]
T : 3
```

- Ou seja, as unidades de número 3, 1 e 4 estão disponíveis e as unidades 2 e 5 estão sendo usadas.



UFV

Caracterização do Problema

- Suponha que 2 processos concorrentes chamem os procedimentos de requisição e liberação
  - Trecho do Processo 1:

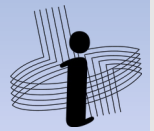
```
requisita(K);
```

  - Trecho do Processo 2:

```
libera(5);
```

  - Suponha que a ordem de execução seja:

```
T = T - 1; //passo 1 de requisita(K). T = 2
R[T] = 5;  //passo 1 de libera(5).    R[2] = 5
T = T + 1; //passo 2 de libera(5).    T = 3
K = R[T];  //passo 2 de requisita(K). K = 2
```



UFV

Caracterização do Problema

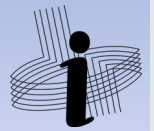
- Antes da execução tínhamos:

```
R : [3, 1, 4, 2, 1]
T : 3
```

- Após a execução teríamos:

```
R : [3, 1, 5, 2, 1]
T : 3
```

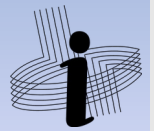
  - A unidade 2 que já estava em uso por um outro processo foi alocada para o processo que chamou requisita(K)!!!
  - A unidade 4 sumiu do sistema!!!
- Este exemplo ilustra o conceito de corrida crítica



UFV

O conceito de região crítica

- Região crítica ou seção crítica é um trecho de código onde uma condição de corrida crítica pode ocorrer
- As variáveis compartilhadas deve estar em uma seção crítica
- Acesso com exclusão mútua

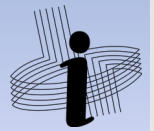


UFV

Exclusão Mútua para 2 *threads*

- Soluções para 2 *threads*
  - Uso de variáveis globais apenas
  - Válidas para um ou mais processadores
- Processos são cíclicos alternando a execução de trechos de acesso à região crítica e à região não crítica

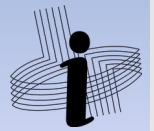
```
while (True) {
    // ...
    // Protocolo de entrada;
    // Região Crítica;
    // Protocolo de Saída;
    // Região não crítica;
    // ...
}
```



UFV

Exclusão Mútua para 2 *threads*

- O protocolo de entrada é implementado com a técnica de espera ocupada (*busy wait*)
- Propriedades dos Algoritmos de Seção Crítica
  - 1) Garantia de Exclusão Mútua
  - 2) Ausência de Bloqueio (*Deadlock*)
  - 3) Ausência de atraso desnecessário
  - 4) Entrada eventual



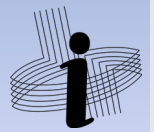
UFV

Exclusão Mútua para 2 *threads*

- Tentativa 1

```
//Variável global
bool em_uso = false;

//Código das threads
void t() {
    ...
    while (em_uso);
    em_uso = true;
    //REGIÃO CRÍTICA
    em_uso = false;
    ...
}
```

Não garante a exclusão Mútua!!
As duas threads podem testar em_uso antes que uma delas faça a atribuição. Ambas passariam pelo protocolo de entrada.



UFV

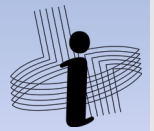
Exclusão Mútua para 2 *threads*

- Tentativa 2

```
//Variável global
int vez = 1;

//Código das threads
void t(int eu, int outro) {
    ...
    while (vez != eu);
    //REGIÃO CRÍTICA
    vez = outro;
    ...
}


/*
'eu' e 'outro' são identificadores.
Para P1, eu=1 e outro=2
Para P2, eu=2 e outro=1
*/
```

Garante a exclusão Mútua!!

Porém obriga a entrada alternada dos processos na Região Crítica (não garante entrada eventual). Se P2 chegar primeiro no protocolo de entrada, ele não entrará (atraso desnecessário). P1 será sempre o primeiro processo a entrar na RC.



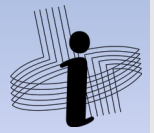
UFV

Exclusão Mútua para 2 *threads*

- Tentativa 3

```
//Variável global
bool quer[2] = {false, false};

//Código das threads
void t(int eu, int outro) {
    ...
    while (quer[outro]);
    quer[eu] = true;
    //REGIÃO CRÍTICA
    quer[eu] = false;
    ...
}

/*
'eu' e 'outro' são identificadores.
Para P1, eu=0 e outro=1
Para P2, eu=1 e outro=0
*/
```

Não garante a exclusão Mútua!!
Problema similar ao da tentativa número 1.

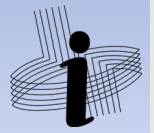


Exclusão Mútua para 2 *threads*

- Tentativa 4

```
//Variável global
bool quer[2] = {false, false};

//Código das threads
void t(int eu, int outro) {
    ...
    quer[eu] = true;
    while (quer[outro]);
    //REGIÃO CRÍTICA
    quer[eu] = false;
    ...
}

/*
'eu' e 'outro' são identificadores.
Para P1, eu=0 e outro=1
Para P2, eu=1 e outro=0
*/
```

Garante a exclusão Mútua!!
Porém, pode gerar um loop eterno (deadlock) e nenhuma das duas threads conseguiria entrar na Região Crítica.

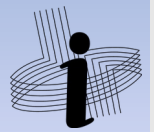


Exclusão Mútua para 2 *threads*

- Tentativa 5

```
//Variável global
bool quer[2] = {false, false};

//Código das threads
void t(int eu, int outro){
    ...
    quer[eu] = true;
    while(quer[outro]){
        quer[eu] = false;
        quer[eu] = true;
    }
    //REGIÃO CRÍTICA
    quer[eu] = false;
    ...
}
```

Garante a exclusão Mútua!!
Porém, pode acontecer de uma thread ficar passando a vez para a outra durante um longo tempo. Por isso, este algoritmo (assim como os 4 anteriores) está incorreto!!!



UFV

Solução de Dekker

- Proposto por T. Dekker em 1965

```
//Variáveis globais
bool quer[2] = {false, false};
int vez = 0;

//Código das threads
void t(int eu, int outro){
    ...
    quer[eu] = true;
    while(quer[outro]) {
        if(vez == eu) continue;
        quer[eu] = false;
        while(true)
            if(vez == eu) {
                quer[eu] = true;
                break;
            }
    }
    //REGIÃO CRÍTICA
    quer[eu] = false; vez = outro;
}
```

Garante a exclusão Mútua!!

Ambas chegam juntas ao PE. Fazem quer[0]=quer[1]=true. A thread cujo número é igual ao valor da variável *vez* volta ao início do PE, enquanto a outra faz quer[eu] = false e fica presa no loop, já que seu número é diferente de vez.

Na próxima tentativa, a thread cujo número é igual ao da variável *vez*, encontrará o valor falso para o teste quer[outro] e entrará na RC. Quando a thread sai da RC, ela tira a outra thread do *loop* ao fazer *vez = outro*.

Solução de Dekker

- Proposto por T. Dekker em 1965

```
//Variáveis globais
bool quer[2] = {false, false};
int vez = 0;

//Código das threads
void t(int eu, int outro){
    ...
    quer[eu] = true;
    while(quer[outro]) {
        if(vez == eu) continue;
        quer[eu] = false;
        while(true)
            if(vez == eu) {
                quer[eu] = true;
                break;
            }
    }
    //REGIÃO CRÍTICA
    quer[eu] = false; vez = outro;
}
```

Garante a ausência de Bloqueio

Conforme mostrado anteriormente, em caso de disputa, a thread cujo número seja igual ao valor da variável vez entrará na RC. Logo, pelo menos uma das threads entrará na RC.

Não possui atraso desnecessário

Se uma das threads chegar sozinha ao PE ela entrará imediatamente, pois o teste quer[outro] será falso.

Garante a entrada eventual

Muito embora a thread "passe a vez" para a outra ao sair da RC, é possível que a thread que deixou a RC entre novamente. Não existe uma alternância obrigatória entre P1 e P2. Logo a propriedade de entrada eventual existe no algoritmo de Dekker.

Solução de Peterson

- Proposto por G. Peterson na década de 80.

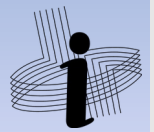
```
//Variáveis globais
bool quer[2] = {false, false};
int vez = 0;

//Código das threads
void t(int eu, int outro){
    ...
    quer[eu] = true;
    vez = outro;
    while(quer[outro] and vez==outro);
    //REGIÃO CRÍTICA
    quer[eu] = false;
    ...
}
```

Garante a exclusão Mútua!
Quando P1 e P2 chegam "juntos" ao PE, a que executar por último comando *vez=outro* ficará presa no loop. A outra entrará na RC.

Garante ausência de bloqueio

Não possui atraso desnecessário!
Se uma thread chegar sozinha, o teste not quer[outro] determina a sua entrada na RC.

Não garante a entrada eventual!
Em caso de disputa, a thread que ganha a entrada na RC não consegue entrar mais de uma vez consecutivamente.

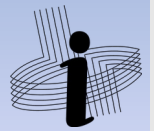


UFV

Solução de Peterson (versão 2)

- Solução igual à anterior, mudando nomes das variáveis (usada na solução para n *threads*)

```
//Variáveis globais
bool quer[2] = {false, false};
int ultimo;

//Código das threads
void t(int eu, int outro){
    ...
    quer[eu] = true;
    ultimo = eu;
    while(quer[outro] and ultimo==eu);
    //REGIÃO CRÍTICA
    quer[eu] = false;
    ...
}
```

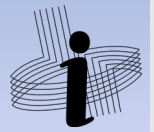


Solução com *"test and set"*

- Instrução especial para ler, testar e atualizar um valor de uma variável usando uma única instrução executada de forma atômica

```
bool TS(bool &x) {
    bool result=x;
    x=false;
    return result;
}
```

- Recurso disponível na grande maioria dos processadores atuais
  - Processador pode atribuir uma *flag busy* em um espaço na memória para execução da instrução, evitando troca de contexto



UFV

Solução com "*test and set*"

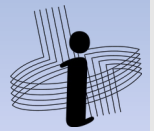
- Exemplo de uso

```
// Variáveis globais
bool livre = true;

//Código das threads
void t(){
    ...
    while(!TS(livre));
    //REGIÃO CRÍTICA
    livre = true;
    ...
}
```

Garante as 4 propriedades!

Pode ser usada com mais de 2 threads

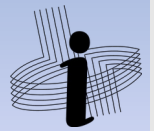
Tem um grande overhead devido à quantidade de chamadas a esta instrução privilegiada



UFV

Exclusão Mútua com n *threads*

- A solução para o problema da exclusão mútua com N *threads* é considerada correta se atende aos seguintes requisitos
  - Garantia de Exclusão Mútua
  - Garantia no progresso para as threads (ausência de *deadlock*)
  - Garantia de tempo de espera limitado (ausência de *starvation*)



UFV

Algoritmo de Dijkstra (1965)

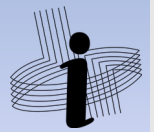
```
//Variáveis globais
bool quer[n]={}, dentro[n]={}; //inicializados com false
int vez = 0;

//Código das threads
void t(int id){
    quer[id]=true;
    do{
        dentro[id]=false;
        do{
            if(!quer[vez]) vez=id;
        } while(vez != id);
        dentro[id]=true;
        bool voltar=false;
        for(int k=0; k<n; ++k) {
            if(k==id) continue;
            if(dentro[k]) voltar=true;
        }
    } while(voltar);
    //REGIÃO CRÍTICA
    quer[id] = false; dentro[id] = false;
}
```

Processo id só entra na RC após fazer *dentro[id]=true* e encontrar as demais posições do vetor dentro igual a false.

Garantia de exclusão mútua!

Ausência de deadlock!

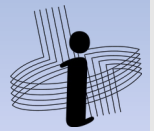
Sem garantia de tempo de espera limitado



UFV

Algoritmo de Eisenberg e McGuire

- Apresentado em 1972
- Similar ao de Dijkstra, corrigindo o problema da espera limitada
  - Ao sair da RC passa a vez para a primeira thread (em ordem cíclica) que tem quer[k]=true
  - Assim, uma thread que faz quer[id]=true esperará no máximo n-1 entrarem na RC
  - Quando nenhuma thread deseja entrar na RC a vez continua com a que está saindo



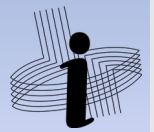
UFV

Algoritmo de Eisenberg e McGuire

Se a thread "*vez*" não quer, a thread *id* verifica se existe outra thread entre *vez* e *id* que quer

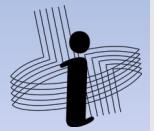
Neste ponto, a exclusão mútua já foi garantida, mas é dada uma chance à thread que tem a vez

```
//Variáveis globais
bool quer[n]={}, dentro[n]={};
int vez = 0;

//Código das threads
void t(int id){
    ...
    int k;
    quer[id]=true;
    while(true){
        dentro[id]=false;
        k=vez;
        do {
            if(!quer[k]) k=(k+1)%n;
            else k=vez;
        } while(k!=id);
        dentro[id]=true;
        bool volta=false;
        for(k=0; k<n; ++k) {
            if(k==id) continue;
            if(dentro[k]) volta=true;
        }
        if(volta) continue;
        if(vez!=id && quer[vez])
            continue;
        vez=id;
        break;
    }
    //REGIÃO CRÍTICA
    k=(vez+1)%n;
    while(!quer[k])
       k=(k+1)%n;
    vez=k;
    quer[id]=false;
    dentro[id]=false;
    ...
}
```



Algoritmo de Lamport

- Apresentado em 1972
- Conhecido como algoritmo do ticket
  - Usado em comércios em geral: atendimento pela ordem de chegada (retirada de um número)
  - Antes de entrar na RC a thread compara o seu número com o de todas as outras threads
  - Primeiro loop: espera outra thread pegar seu ticket. Segundo loop: determina se a thread *j* está disputando a RC e compara número de ticket e número da thread (se necessário)
- Garante exclusão mútua, progresso de todas as threads e espera limitada



UFV

Algoritmo de Lamport

```
//Variáveis globais
bool pegando[n] = {};  //inicializado com false
int  ticket[n] = {};   //inicializado com 0

//Código das threads
void t(int id){
    ...
    pegando[id]=true;
    ticket[id]=max(ticket,n)+1;
    pegando[id]=false;
    for(int j=0; j<n; ++j){
        if(id==j) continue;
        while(pegando[j]);
        while(ticket[j]!=0 &&
              !menor(ticket[id],ticket[j],id,j));
    }
    //REGIÃO CRÍTICA
    ticket[id]=0;
    ...
}
```

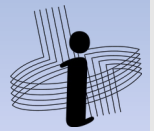
```
int max(int *t,n) {
    return *max_element(t,t+n);
}

bool menor(a,b,c,d) {
    return(a<b)||(a==b)&&(c<d);
}
```

Espera para ter certeza de que uma thread com *id* menor não vai pegar o mesmo número

No caso de empate no número do ticket, o desempate é feito pelo número da thread.

Algoritmo de Peterson

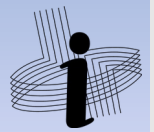
- Proposto em 1981
- A cada estágio *j* uma thread (a última a atualizar a posição *j* do vetor *ultimo*) fica presa e as demais avançam
- Exclusão mútua
  - Primeiro estágio deixa passar no máximo n-1 threads
  - Segundo estágio deixa passar no máximo n-2 threads
  - Último estágio deixa passar apenas 1 thread
- Garante progresso e espera limitada



UFV

Algoritmo de Peterson

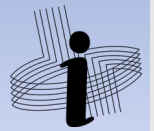
```
//Variáveis globais
int estagio[n];   //estágio de cada thread iniciado com -1 (omitido)
int ultimo[n-1];  //última thread a entrar em cada estágio

//Código das threads
void t(int id){
    ...
    for(int j=0; j<n-1; ++j){
        estagio[id]=j;
        ultimo[j]=id;
        for(k=0; k<n; ++k) {
            if(k==id) continue;
            while(estagio[id]<=estagio[k] && ultimo[j]==id);
        }
    }
    //REGIÃO CRÍTICA
    estagio[id] = -1;
    ...
}
```



UFV

Algoritmo de Block e Woo

- Proposto em 1990
- Otimização do algoritmo de Peterson
- Quando a thread verifica que está na frente de todos os demais ela entra na RC imediatamente
  - Cada thread informa o estágio atual em uma posição do vetor estágio
  - A thread avança um estágio quando ele deixa de ser o último de seu estágio
  - Se o seu estágio é igual ao número de threads que está tentando entrar na RC, ela está habilitado a entrar na RC

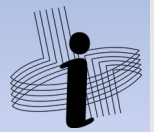


UFV

Algoritmo de Block e Woo

```
//Variáveis globais
int quer[n] = {};   //sinal de que a thread quer entrar (iniciado com 0)
int ultimo[n-1];    //última thread a entrar em cada estágio

//Código das threads
void t(int id){
    ...
    int estagio=0;
    quer[id]=1;
    do {
        estagio+=1;
        ultimo[estagio]=id;
        while(ultimo[estagio]==id &&
              estagio>soma(quer)-1);
    } while(ultimo[estagio]!=id;
    //REGIÃO CRÍTICA
    quer[id] = 0;
    ...
}
```

```
int soma(int *a,n) {
    int x=0;
    for(int i=0;i<n;++i)
        x+=a[i];
    return x;
}
```



UFV

Analisando o uso de Espera Ocupada

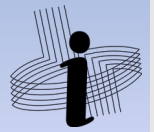
- Ciclos de CPU potencialmente desperdiçados
- Otimização realizada pelo compilador pode gerar resultados errados
  - Padrão do gcc = não otimizar (-O0)
  - Otimização pode ser ativada com a diretiva -O2, por exemplo

```
...
meu_x = calcula(meu_id);//meu_x é variável privada
while (vez != meu_id);
x += meu_x;                 //x é compartilhada
vez=(vez+1)%num_threads;
...
```

Instruções independentes podem ter suas ordens alteradas (2ª com 3ª ou 3ª com 4ª)

  - Erro pode ser evitado informando que outras *threads* têm acesso às variáveis

```
int volatile x;
int volatile vez;
```



UFV