Computação Concorrente (DCC/UFRJ)

Módulo 1 - Semana 4 - Aula 1: Comunicação entre threads via memória compartilhada e sincronização por exclusão mútua

Prof. Silvana Rossetto

Dezembro 2020

Problema

Dada uma sequência de números inteiros positivos, identificar todos os **números primos** e retornar a quantidade encontrada

Função para verificar a primalidade

```
int ehPrimo(long long int n) {
    if (n<=1) return 0;
    if (n==2) return 1;
    if (n%2==0) return 0;
    for (int i=3; i<sqrt(n)+1; i+=2)
        if(n%i==0) return 0;
    return 1;
```

Como dividir essa tarefa entre várias threads?

Solução 1: divisão estática das tarefas

```
long long int sequencia[N];
void * conta_primos_1(void * arg) {
   int id = (int) arg;
   total=0, *ret;
   for (int i=id; i<N; i+=NTHREADS) {
      if(ehPrimo(sequencia[i]))
         total++;
   ret = (int*) malloc(sizeof(int));
   *ret = total;
   pthread_exit((void *)ret);
}
```

Essa solução garante balanceamento de carga?

Solução 1: divisão estática das tarefas

Tempos de execução para N=1000000:

threads/tempo(s)	Sequencial	Concorrente
1 thread	0.525401	0.544142
2 threads	0.525401	**0.542982**
3 threads	0.525401	0.317463
4 threads	0.525401	0.322370

Solução 2: divisão "dinâmica" das tarefas

```
int i_global=0; //variavel compartilhada
void * conta_primos_2(void * arg) {
   int i_local, total=0;
   //!!acessa variavel compartilhada!!
   i_local = i_global; i_global++;
   while(i_local < N) {
      if(ehPrimo(sequencia[i_local]))
         total++;
      //!!acessa variavel compartilhada!!
      i_local = i_global; i_global++;
   }
   //...retorna 'total'
}
```

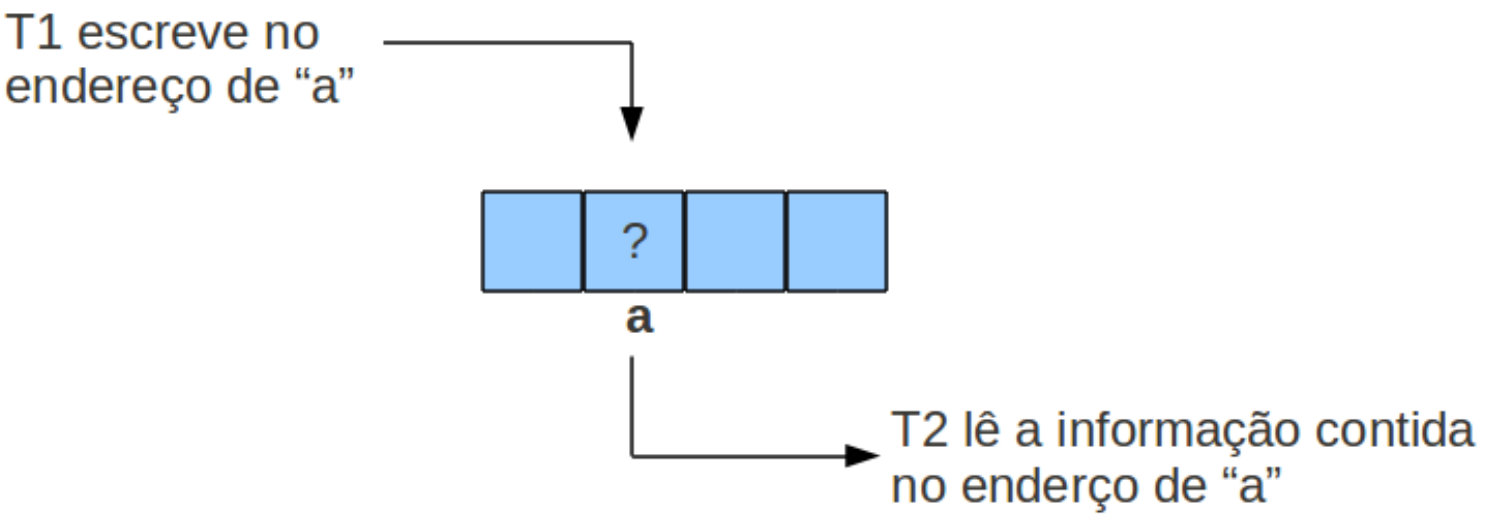
Solução 2: divisão "dinâmica" das tarefas

Tempos de execução para N=1000000:

threads/tempo(s)	Sequencial	Concorrente
1 thread	0.525401	0.544142
2 threads	0.525401	**0.338769**
3 threads	0.525401	0.312914
4 threads	0.525401	0.308235

Comunicação via memória compartilhada

- Quando uma thread tem um valor para ser comunicado para as demais threads, ela **escreve esse valor na variável compartilhada**
- Quando outra thread precisa saber qual é o valor atual dessa informação, ela **lê o conteúdo atual da variável compartilhada**



Comunicação assíncrona

Com memória compartilhada, a comunicação entre as threads é **assíncrona**: as threads escrevem/lêem valores nas variáveis compartilhadas a qualquer tempo

Concorrência dentro do código de uma aplicação

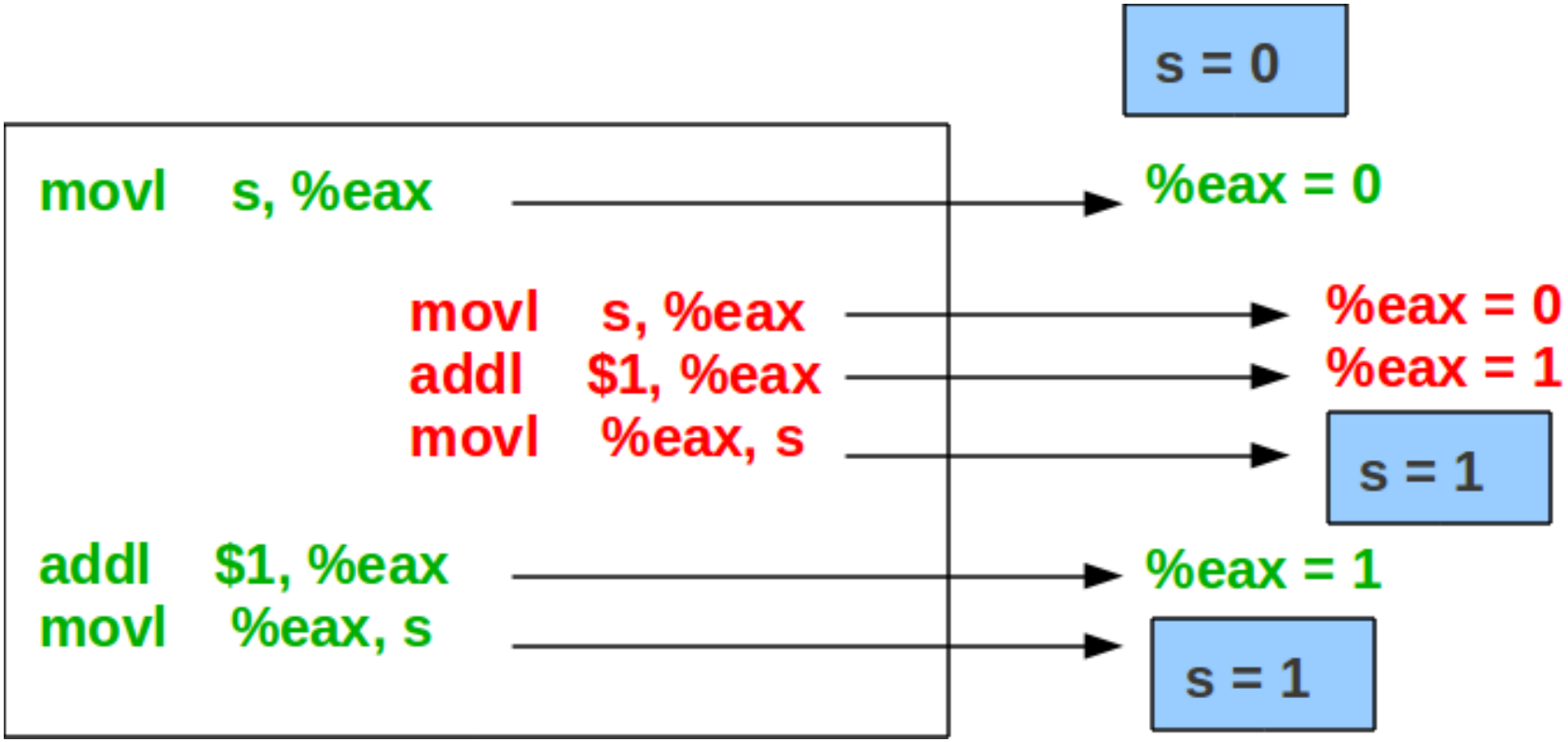
```
int s;

void soma() {
      s++;
}
```

```
.comm   s,4,4
soma: (...)
      movl    s, %eax

      addl    $1, %eax

      movl    %eax, s
      (...)
```

Interrupção do tempo

Interrupção do tempo

Condição de corrida dentro do código de uma aplicação



Exercício

```
int x=10 //variável global
```

```
T1: while (1) {
        x = x - 1;
        x = x + 1;
        if (x != 10)
          printf("x is %d",x);
      }
```

```
T2: while (1) {
        x = x - 1;
        x = x + 1;
        if ( x != 10)
          printf("x is %d",x);
      }
```

- O que espera-se que seja impresso na tela após cada loop das threads?
- Pode ocorrer de **"x is 10"** ser impresso?
- Pode ocorrer de **"x is 9"** ser impresso?
- Pode ocorrer de **"x is 11"** ser impresso?

Condição de corrida

Quando o resultado da computação depende da ordem em que as diferentes linhas de execução acessam uma variável comum, chamamos de condição de corrida

Condição de corrida

- Nem toda condição de corrida é “ruim”, i.e., algumas vezes qualquer saída do programa é aceitável
- Temos que nos preocupar em resolver as condições de corrida “ruins”: aquelas que fazem o programa produzir resultados incorretos!

Seção crítica do código e sincronização

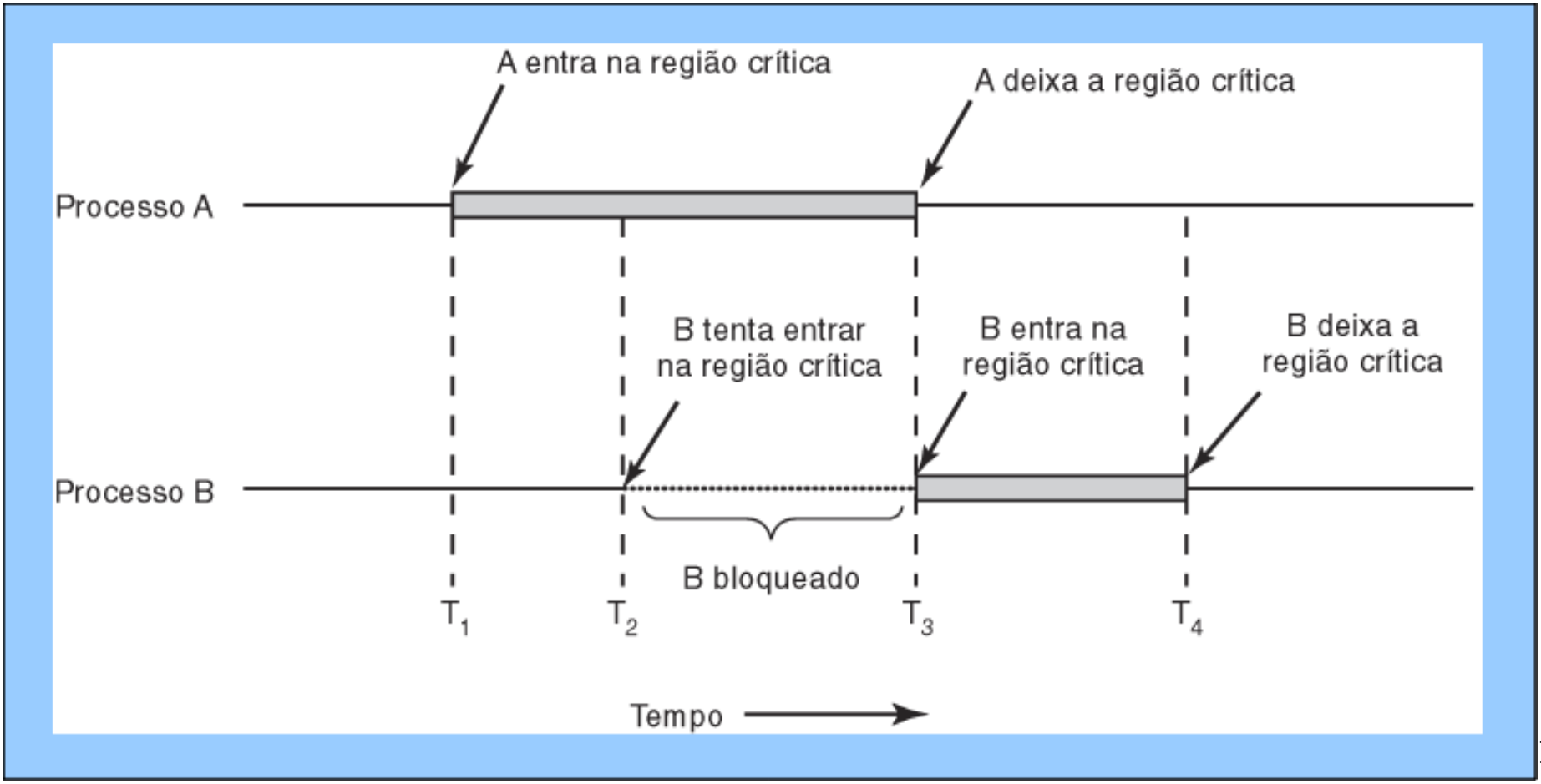
- **Seção crítica do código** refere-se ao trecho do código onde uma variável compartilhada por mais de uma thread é acessada (leitura/escrita)
- **Sincronização** refere-se a qualquer mecanismo que permite ao programador controlar a ordem relativa na qual as operações ocorrem em diferentes threads
- Sincronização por exclusão mútua visa garantir que as **seções críticas** do código não sejam executados ao mesmo tempo por mais de uma thread (impede "condições de corrida ruins")

Sincronização por exclusão mútua

- As **seções críticas de código** devem ser transformadas em **AÇÕES ATÔMICAS**

Assim a execução de uma seção crítica NÃO ocorre simultaneamente com outra seção crítica que referencia a mesma variável

Controle de acesso à seção crítica



[1] *Fonte: Pearson*

Seções de entrada e saída da seção crítica

```
while (true) {
    requisita a entrada na seção crítica  //seção de entrada
    executa a seção crítica               //seção crítica
    sai da seção crítica                  //seção de saída
    executa fora da seção crítica
 }
```

Condições para implementar a exclusão mútua

1. **Apenas uma thread na seção crítica a cada instante**
2. Nenhuma suposição sobre velocidade das threads
3. Nenhuma thread fora da seção crítica pode impedir outra thread de continuar
4. Nenhuma thread deve esperar indefinidamente para executar a sua seção crítica

Espera ocupada versus escalonamento

Há duas abordagens básicas para implementar a sincronização por exclusão mútua:

1. **por espera ocupada**: a thread fica continuamente testando o valor de uma determinada variável até que esse valor lhe permita executar a sua seção crítica com exclusividade
2. **por escalonamento**: alternativa mais usual, nas formas **locks**, **semáforos** e **monitores**

Exclusão mútua com *locks*

- Um **lock** é uma **variável de sincronização** para resolver o problema de **exclusão mútua** no acesso a **variáveis/recursos compartilhados**

T1:	**T2:**	**T3:**
L.lock();	L.lock();	L.lock();
//seção crítica	//seção crítica	//seção crítica
L.unlock();	L.unlock();	L.unlock();

Propriedades dos *locks*

O **lock (L) possui uma thread proprietária**:

1. A thread requisita a posse de um *lock* L com a operação `L.lock()`;
2. A thread que executa **L.lock()** torna-se a proprietária do *lock* L se ele estiver livre, caso contrário a thread é bloqueada
3. A thread libera sua posse sobre o *lock* L executando **L.unlock**
4. A thread que já possui o lock **L** e executa **L.lock()** novamente não é bloqueada (mas deve executar **L.unlock()** o mesmo número de vezes que **L.lock()**) (requer propriedade de **lock recursivo**)

Lock na biblioteca Pthreads

- A biblioteca Pthreads oferece o mecanismo de *lock* através de variáveis especiais do tipo **pthread_mutex_t**
- Por padrão, Pthreads implementa **locks não-recursivos** (uma thread não deve tentar alocar novamente um *lock* que já possui)
- Para tornar o *lock* recursivo é preciso mudar suas propriedades básicas

Exemplo de lock não-recursivo em Pthreads/C

```
pthread_mutex_t mutex =
                    PTHREAD_MUTEX_INITIALIZER;
   //ou: pthread_mutex_init(&mutex, NULL);
...
//trecho de código nas threads

//entrada na secao critica
pthread_mutex_lock(&mutex);
   ... //secao critica
//saida da secao critica
pthread_mutex_unlock(&mutex);
...
pthread_mutex_destroy(&mutex);
```

voltando à solução 2: divisão "dinâmica" das tarefas

```
int i_global=0; pthread_mutex_t bastao;
void * conta_primos_2(void * arg) {
   i_local, total=0, ...

   pthread_mutex_lock(&bastao);
   i_local = i_global; i_global++;
   pthread_mutex_unlock(&bastao);

   while(i_local < N) {
      if(ehPrimo(sequencia[i_local])) { total++; }
      pthread_mutex_lock(&bastao);
      i_local = i_global; i_global++;
      pthread_mutex_unlock(&bastao);
   }
   //retorno de 'total'...
}
```

Exercícios

1. O que é **seção crítica** de um código?
2. O que caracteriza um programa com **condição de corrida**?
3. O que é "condição de corrida ruim"?
4. O que é **sincronização por exclusão mútua**?

Referências bibliográficas

- *Computer Systems - A Programmer's Perspective* (Cap. 12)
- *An Introduction to Parallel Programming*, Peter Pacheco, Morgan Kaufmann, 2011