Computação Concorrente (DCC/UFRJ)

Aula 5: Comunicação entre threads via memória compartilhada e sincronização por exclusão mútua

Prof. Silvana Rossetto

10 de setembro de 2019

Observações importantes último lab...

$$\pi = 4\left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots\right)$$

- diferentes estratégias para dividir a soma dos elementos da série entre as threads
- escolher a estratégia que minimize o erro numérico

Exercício

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

Como dividir essa tarefa entre várias threads?

Exercício: divisão estática das tarefas

```
long long int sequencia[N];
void * conta_primos_1(void * args) {
   int id = *(int*)args;
   int i, total=0, *ret;
   for (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?

Exercício: divisão estática das tarefas

(*mostrar execução no computador com 1, 2, 3 e 4 threads...*)

Exercício: divisão "dinâmica" das tarefas

```
int i_global=0; ...
void * conta_primos_2(void * args) {
   int i_local, total=0, *ret;

   i_local = i_global; i_global++;
   while(i_local < N) {
      if(ehPrimo(sequencia[i_local]))
         total++;
      i_local = i_global; i_global++;
   }
   ret = (int*) malloc(sizeof(int));
   *ret = total;
   pthread_exit((void *)ret);
}
```

Qual é o problema (erro) desse código?

Comunicação entre threads

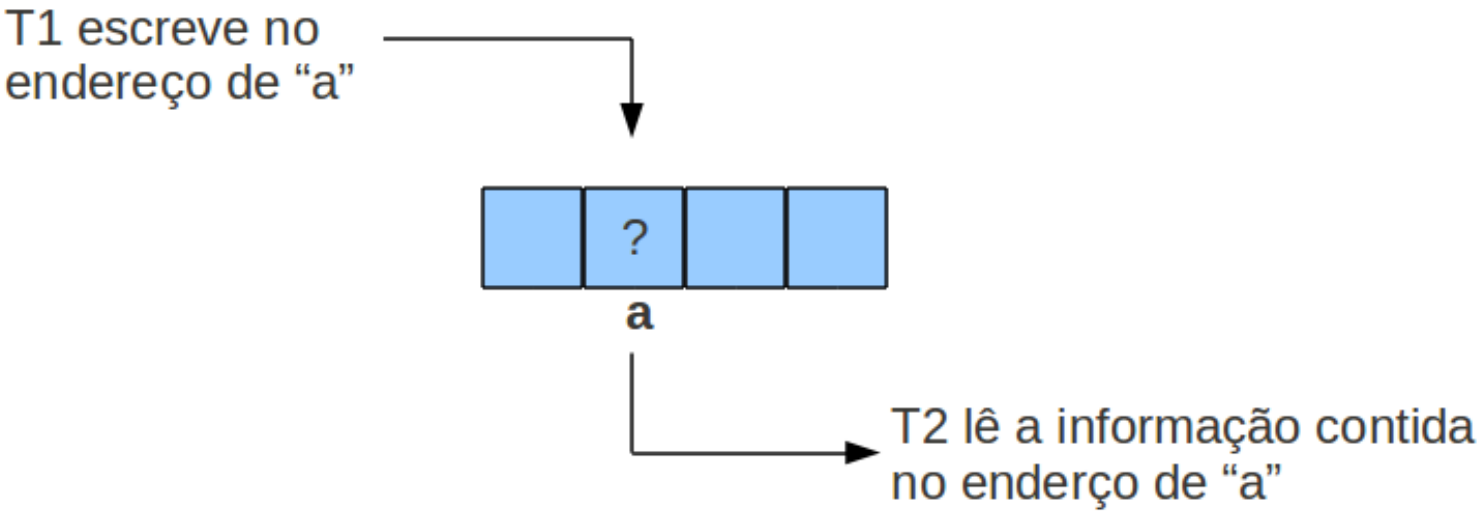
Nesse problema, precisamos que as threads da aplicação **troquem informação entre si**: para saber qual é o próximo número a ser processado (?)

A facilidade de **espaço de endereçamento físico único** pode ser usada para implementar comunicação entre threads via **memória compartilhada**

Comunicação via memória compartilhada

comunicação assíncrona

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



Programação concorrente sem erros

Para escrever **programas concorrentes corretos** é preciso ter um entendimento claro do que significa "compartilhar variáveis e como isso funciona"!

Modelo de memória das threads

Threads de um mesmo processo/aplicação não podem ler/escrever os registradores de outras threads
...mas podem acessar as mesmas localizações de memória compartilhada

1. variáveis **globais**: qualquer variável declarada fora de uma função, possui apenas uma instância no programa, todas as threads podem ler e escrever
2. variáveis **locais automáticas**: declaradas dentro de uma função, são armazenadas na pilha de cada thread
3. variáveis **locais estáticas (em C)**: declaradas dentro de uma função com o atributo **static**, possui apenas uma instância no programa, todas as threads podem ler e escrever se tiverem acesso à função que define a variável

Variável compartilhada

Uma variável 'v' é compartilhada quando (ao menos) uma instância dessa variável é referenciada por mais de uma thread!

Concorrência dentro do código de uma aplicação

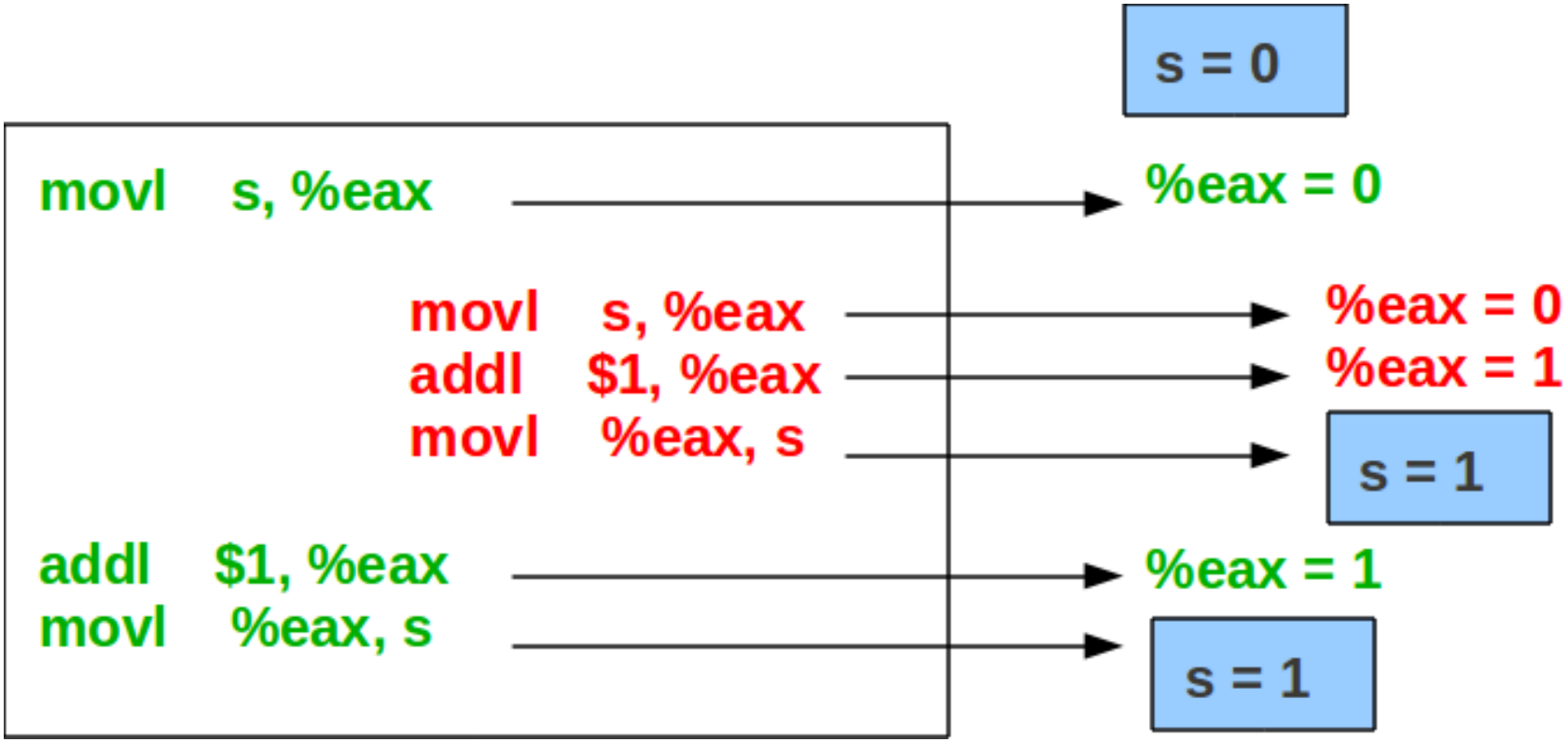
```
int s;

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

Seção crítica do código refere-se ao trecho do código onde uma variável compartilhada por mais de uma thread é acessada (leitura/escrita)

Sincronização

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

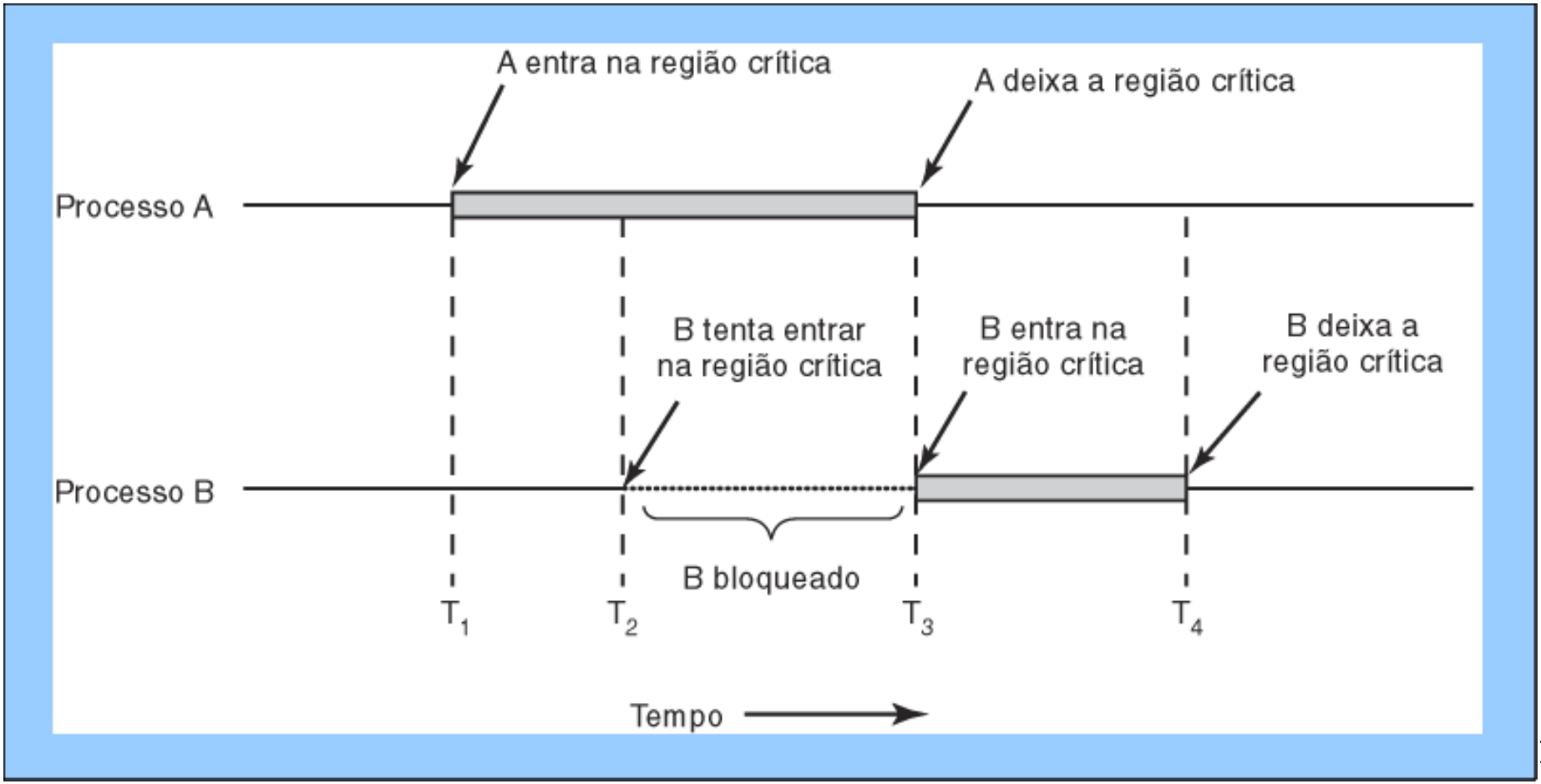
- impede "condições de corrida ruins"
- *ex.: uma thread não pode ler o valor de uma variável enquanto outra thread estiver alterando essa variável*

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

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. Uma thread requisita a posse de um *lock* L executando a operação `L.lock();`
2. Uma thread que executa **L.lock()** torna-se a proprietária do *lock* quando nenhuma outra thread possui o *lock*, caso contrário a thread é bloqueada
3. Uma thread libera sua posse sobre o *lock* executando **L.unlock** (se a thread não possui o lock a operação retorna com erro)
4. Uma thread que já possua o lock **L** e executa **L.lock()** novamente não é bloqueada (mas deve executar **L.unlock()** o mesmo número de vezes que **L.lock()** para o controle passar para outra thread) (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
pthread_mutex_lock(&mutex); //entrada na secao critica
  ... //secao critica
pthread_mutex_unlock(&mutex); //saida da secao critica
...
pthread_mutex_destroy(&mutex);
```

voltando ao exercício inicial: divisão "dinâmica" das tarefas

```
int i_global=0; pthread_mutex_t bastao;
void * conta_primos_2(void * args) {
   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'...
}
```

Exercício: divisão "dinâmica" das tarefas

(*mostrar execução no computador com 1, 2 e 3 threads...*)

Exercício

Quais variáveis são compartilhadas entre as threads? Há condição de corrida nesse código?

```
#define N 2
char **ptr; int id[N];
void *thread (void *vargp) {
 int myid = *(int*) vargp;
 static int cnt = 0;
 printf("[%d]: %s (cnt=%d)\n", myid, prt[myid], ++cnt);
}

int main() {
 int i; pthread_t tid[N];
 char *msgs[N] = {"hello from foo", "hello from bar"};
 ptr = msgs;
 for (i=0; i<N; i++)
  pthread_create(&tid, NULL, thread, (void *)&id[i]);
}
```

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)
- *Programming Language Pragmatics*, M.L.Scott, Morgan-Kaufmann, ed. 2, 2006
- *Modern Multithreading*, Carver e Tai, Wiley, 2006
- *An Introduction to Parallel Programming*, Peter Pacheco, Morgan Kaufmann, 2011