Computação Concorrente (DCC/UFRJ)

Aula 7: Problemas clássicos de concorrência usando locks e variáveis de condição

Prof. Silvana Rossetto

8 de outubro de 2019

Barreiras

Sincronização com barreira

- Vários problemas computacionais são resolvidos usando **algoritmos iterativos que sucessivamente computam aproximações melhores para uma resposta procurada** (*ex., resolução de sistemas de equações lineares usando o método de Jacobi*)
- Esses algoritmos manipulam um vetor de valores e a cada iteração executam a mesma computação sobre todos os elementos do vetor, melhorando os valores anteriores

- É possível usar várias threads para computar partes disjuntas da solução de forma concorrente/paralela
- **Um requisito é que cada iteração depende da anterior, então as threads devem aguardar a próxima iteração**

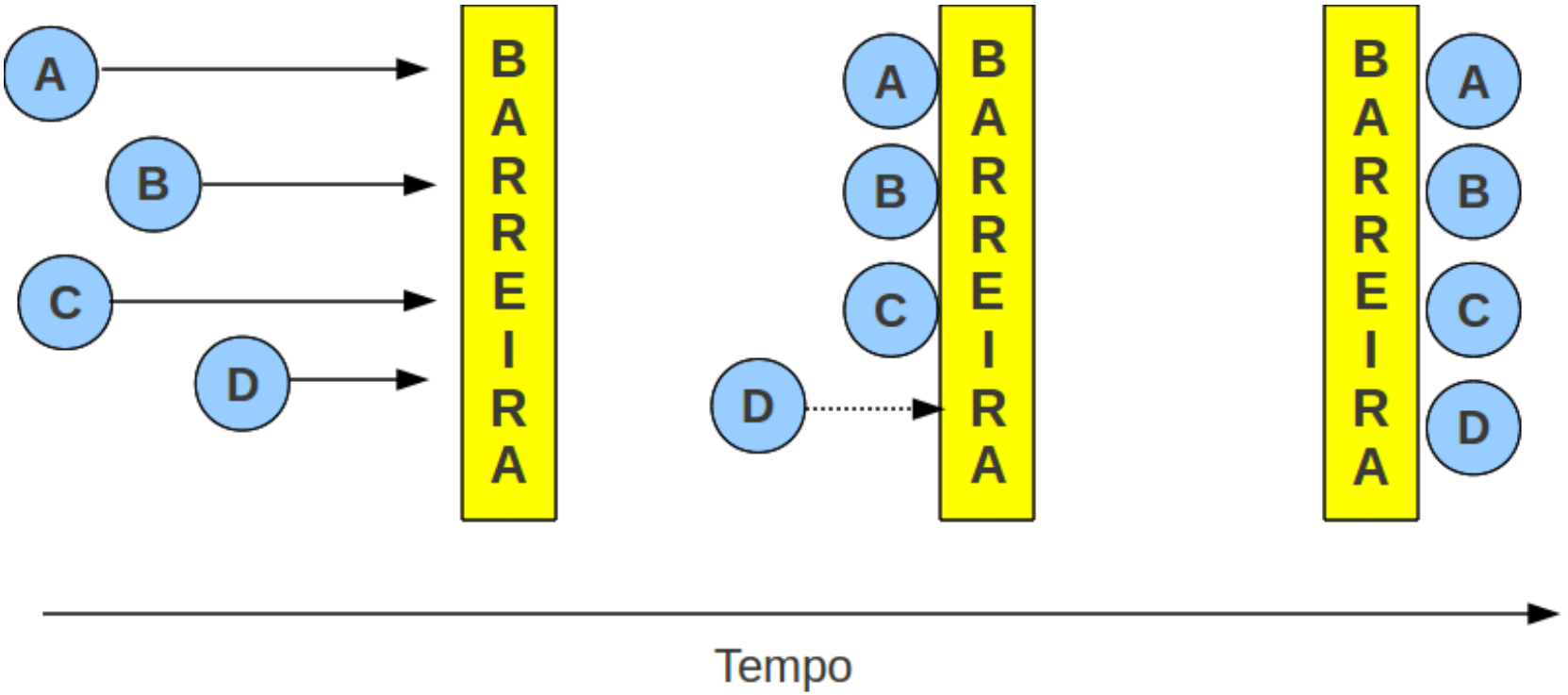
Sincronização por barreira

Para garantir que as threads trabalhem sempre em fase (na mesma iteração) é necessário usar um tipo de sincronização chamada **sincronização por barreira**

Barreira

Um tipo de sincronização coletiva que suspende a execução das threads de um aplicação em um dado ponto do código e somente permite que as threads prossigam quando todas elas tiverem chegado naquele ponto

Exemplo sincronização por barreira



Implementação de sincronização por barreira

1. Uma maneira simples de implementar uma barreira é usar um **contador que é inicializado com o número total de threads envolvidas**
2. Cada thread decrementa o contador após alcançar a barreira e então se bloqueia esperando o contador chegar a zero
3. Quando o contador chega a zero todas as threads são desbloqueadas

Exercício

- Implemente uma função **void barreira(int nthreads)** para implementar sincronização coletiva
- O parâmetro `nthreads` informa o número total de threads que devem participar da barreira
- Todas as threads deverão chamar essa função no ponto do código onde a sincronização por barreira é requerida
- Dica: use **locks** e **variáveis de condição**

Exemplo de implementação de barreira

```
int threads=0;
pthread_mutex_t mutex;
pthread_cond_t cond_bar;

void barreira(int nthreads) {
   pthread_mutex_lock(&mutex);
   threads++;
   if (threads < nthreads) {
      pthread_cond_wait(&cond_bar, &mutex);
   } else {
      threads=0;
      pthread_cond_broadcast(&cond_bar);
   }
   pthread_mutex_unlock(&mutex);
}
```

O problema dos leitores e escritores

O problema dos leitores e escritores

Definição

- Uma área de dados (ex., arquivo, bloco da memória, tabela de uma banco de dados) é compartilhada entre diferentes threads
- As **threads leitoras** apenas lêem o conteúdo da área de dados
- As **threads escritoras** apenas escrevem conteúdo na área de dados

exemplo de aplicação

em um sistema de reservas de passagens aéreas, um grande número de usuários pode inspecionar concorrentemente os assentos disponíveis, mas um usuário que está reservando um assento deve ter acesso exclusivo ao controle desse assento

O problema dos leitores e escritores

Condições do problema

1. Os leitores podem ler simultaneamente uma região de dados compartilhada
2. Apenas um escritor pode escrever a cada instante em uma região de dados compartilhada
3. Se um escritor está escrevendo, nenhum leitor pode ler a mesma região de dados compartilhada

Código das threads

```
void *leitor (void *arg) {
  while(1) {
    EntraLeitura();
    //le algo...
    SaiLeitura();
    //faz outra coisa...
  }
}

void *escritor (void *arg) {
  while(1) {
    EntraEscrita();
    //escreve algo...
    SaiEscrita();
    //faz outra coisa...
  }
}
```

Exercício

Implemente o código das funções:

1. **EntraLeitura**
2. **SaiLeitura**
3. **EntraEscrita**
4. **SaiEscrita**

Funções para leitura

```
int leit=0, escr=0; //globais
void EntraLeitura() {
   pthread_mutex_lock(&mutex);
   while(escr > 0) {
     pthread_cond_wait(&cond_leit, &mutex);
   }
   leit++;
   pthread_mutex_unlock(&mutex);
}

void SaiLeitura() {
   pthread_mutex_lock(&mutex);
   leit--;
   if(leit==0) pthread_cond_signal(&cond_escr);
   pthread_mutex_unlock(&mutex);
}
```

Funções para escrita

```
int leit=0, escr=0; //globais
void EntraEscrita (int id) {
   pthread_mutex_lock(&mutex);
   while((leit>0) || (escr>0)) {
     pthread_cond_wait(&cond_escr, &mutex);
   }
   escr++;
   pthread_mutex_unlock(&mutex);
}

void SaiEscrita (int id) {
   pthread_mutex_lock(&mutex);
   escr--;
   pthread_cond_signal(&cond_escr);
   pthread_cond_broadcast(&cond_leit);
   pthread_mutex_unlock(&mutex);
}
```

Exercício

Considere uma variação do padrão leitores/escritores onde precisamos dar **prioridade para a escrita** (quando um escritor quer escrever, novos leitores não podem começar a ler)

Implemente a nova versão para o código das funções:

1. **EntraLeitura**
2. **SaiLeitura**
3. **EntraEscrita**
4. **SaiEscrita**

Referências bibliográficas

- *An Introduction to Parallel Programming*, Peter Pacheco, Morgan Kaufmann, 2011