Introdução aos Mecanismos de Exclusão Mútua

Profa. Denise Stringhini
ICT-Unifesp

Compartilhamento de memória

- As threads em execução concorrem pelo acesso a todos os recursos disponíveis ao processo.
  - Incluindo as variáveis compartilhadas.
- Problema: garantir que apenas uma das threads acesse um dado compartilhado num determinado instante de tempo.

Compartilhamento de memória

- Exemplo:

```
int x; //variável global compartilhada (seção crítica)

// thread A
a = x;
a = a + 1;
x = a;

// thread B
b = x;
b = b - 1;
x = b;
```

- Considerando uma execução simultânea, as instruções podem ocorrer de forma intercalada...

Compartilhamento de memória

- Exemplo de execução:

Thread	Instrução	x	a	b
A	a = x;	313	313	-
A	a = a+1;	313	314	-
B	b = x;	313	314	313
A	x = a;	314	314	313
B	b = b-1;	314	314	312
B	x = b;	312	314	312

- Um mecanismo de sincronização deve ser empregado para evitar este problema.

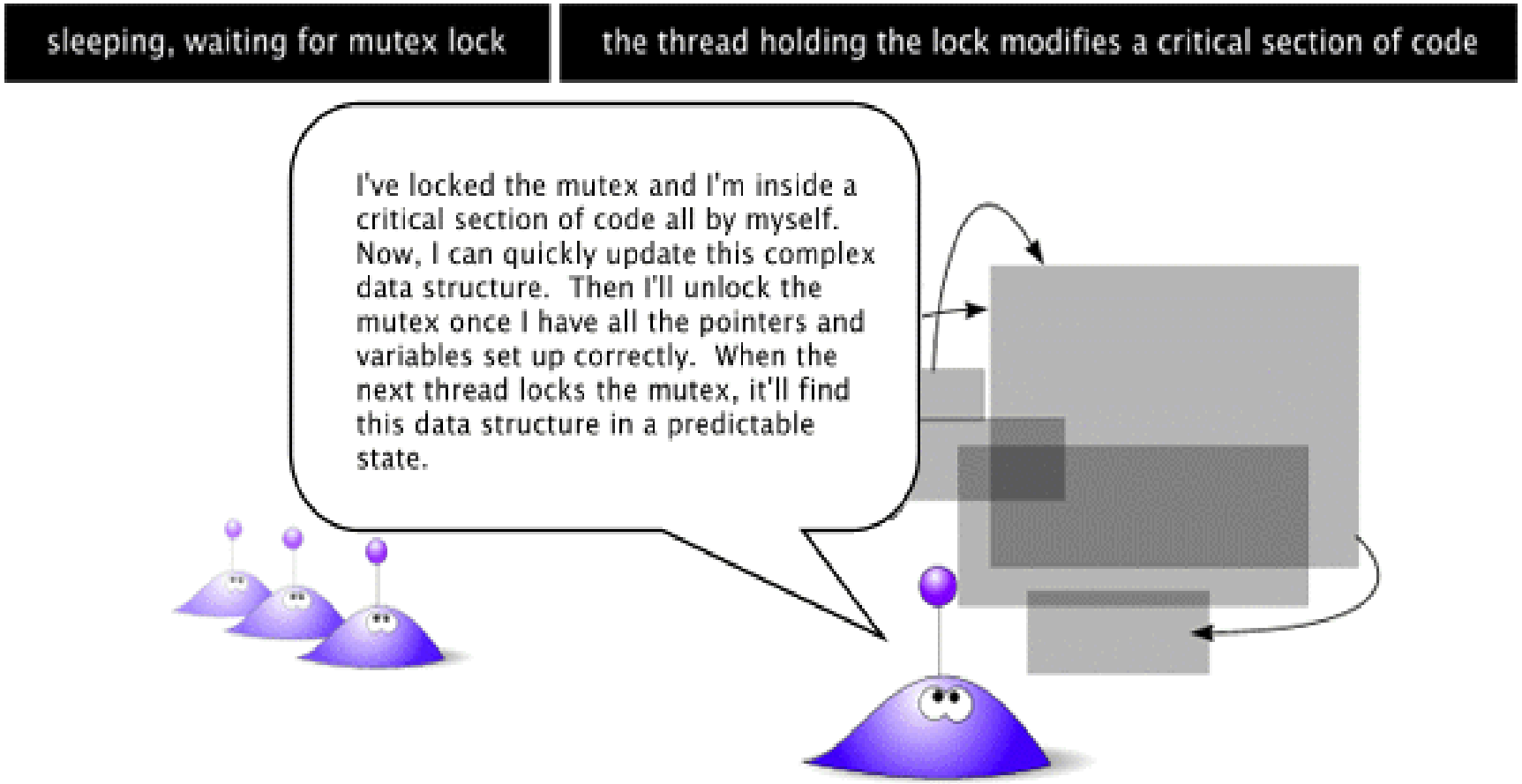
Mecanismos de sincronização

- Mutex
- Variáveis de condição
- Semáforos
- Monitores

Mutex

- Permite que uma thread tenha acesso exclusivo a uma área de dados.
- Operações:
  - *lock*: permite acesso a uma thread e bloqueia as demais (*fecha a "porta"*).
  - *unlock*: libera o acesso às demais threads (*abre a "porta"*).

Mutex



Fonte: www.ibm.com/developerworks/library/l-posix2/

Mutex

- Exemplo: Pthreads

```
int x; //variável global compartilhada (seção crítica)
pthread_mutex_t m;

// thread A                        // thread B
pthread_mutex_lock(&m);            pthread_mutex_lock(&m);
a = x;                             b = x;
a = a + 1;                         b = b - 1;
x = a;                             x = b;
pthread_mutex_unlock(&m);          pthread_mutex_unlock(&m);
```

Variáveis de condição

- Permitem o avanço da thread apenas se uma determinada condição for satisfeita.
- São utilizadas em conjunto com outros mecanismos de sincronização (ex: mutex).
- São implementadas através do recurso de sinais, que evitam que a condição tenha que ser testada em intervalos regulares.

Variáveis de condição

- Operações:
  - wait: bloqueia a thread à espera de um sinal
  - signal: desbloqueia uma thread que esteja bloqueada
  - broadcast: desbloqueia todas as threads

Variáveis de condição

- Exemplo: produtor-consumidor

```
//Área de memória compartilhada
Buffer b;
int cont;
pthread_mutex_t mb;
pthread_cond_t c;

void  produtor(){
 Item it;
 for(;;){
   it = produzItem();
   pthread_mutex_lock(&mb);
   armazenaBuffer(b, it);
   cont++;
   pthread_cond_signal(&c);
   pthread_mutex_unlock(&mb);
 }
}
```

```
void consumidor(){
 Item it;
 for(;;){
   pthread_mutex_lock(&mb);
   while(cont <= 0)
     pthread_cond_wait(&mb, &c);
   it = leBuffer(b);
   cont--;
   pthread_mutex_unlock(&mb);
 }
}
```

Variáveis de condição

- OBS:
  - Os sinais normalmente não são memorizados...
  - Somente as threads que estejam em estado de wait no momento do envio (condição satisfeita) estão aptas a receber o sinal.
  - Caso contrário ele é perdido.

Semáforo

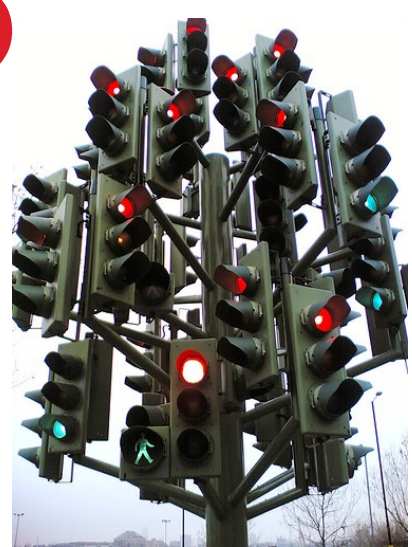
- Permite controlar o avanço de um grupo de threads sobre um trecho de código.
- Mantém um contador que indica o estado do semáforo.
  - Controla a quantidade de threads que tem permissão de avançar e utilizar os recursos compartilhados.
- Operações:
  - P : testa o contador e libera a passagem (ou bloqueia)
  - V : libera o avanço da próxima thread

Semáforo

- Funcionamento:
  - A thread que deseja avançar executa a operação P.
    - O contador é decrementado.
    - A thread tem permissão de passagem se o valor resultante for maior ou igual a zero.
    - Caso contrário é bloqueada e colocada em uma fila.
  - A operação V deve ser executada pela thread que sai da região compartilhada.
    - O que vai liberar a próxima thread e decrementar o contador novamente.

Semáforo

- Semáforo binário
  - É aquele que só assume os valores 0 ou 1, garantindo a exclusão mútua.



Semáforo: Filósofos jantadores (exemplo)

- Cinco filósofos estão sentados ao redor de uma mesa circular e se alternam entre duas atividades: pensar e comer espaguete.
- Necessitam de dois garfos para comer, mas a mesa só possui cinco garfos, cada um localizado entre dois filósofos. Eles concordam em utilizar somente garfos imediatamente à sua esquerda e direita.

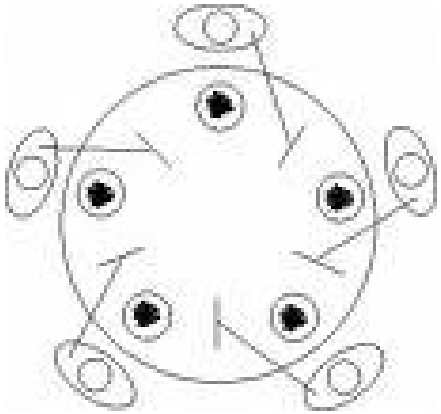
Semáforo: Filósofos jantadores (exemplo)

- O problema consiste em simular o comportamento dos filósofos evitando que passem fome por não conseguirem adquirir ambos os garfos (ex: cada um consegue um garfo e se recusa a desistir dele).
- Caracteriza a alocação concorrente de recursos, onde o *deadlock* deve ser evitado.
- Claramente, dois filósofos vizinhos não conseguem comer ao mesmo tempo e no máximo dois filósofos o conseguirão.

Semáforo: Filósofos jantadores (exemplo)

- O problema está descrito no pseudo-código abaixo, que utiliza a notação de Andrews, 2000:

```
process Filósofo [i = 0 to 4]
    while (true){
    // pensar
    // adquirir garfo
    // comer
    // liberar garfo
    }
}
```



- Uma situação perigosa seria implementar o mesmo código para todos, o que poderia levar ao *deadlock* circular, por exemplo, se todos adquirirem inicialmente o garfo à sua esquerda.

Semáforo: Filósofos jantadores (exemplo)

- A solução a seguir implementa os garfos através de um vetor de semáforos – inicialmente todos estão liberados (sobre a mesa).
- Nesta solução, quatro dos filósofos iniciam pelo garfo à esquerda, com exceção do último filósofo, que adquire inicialmente o garfo da direita.
- Com isso, o círculo é quebrado e pelo menos um dos filósofos estará apto a conseguir dois garfos.

Semáforo: Filósofos jantadores (exemplo)

```
sem fork[5] = {1, 1, 1, 1, 1};
process Filósofo [i = 0 to 3]{
    while (true){
        P(fork[i]);
P(fork[i+1]);
        //come
        V(fork[i]);
V(fork[i+1]);
        //pensa
    }
}
```

```
process Filósofo [4]{
    while (true){
        P(fork[0]); P(fork[4]);
        //come
        V(fork[0]); V(fork[4]);
        //pensa
    }
}
```

Exercício: escreva um código SPMD para este exemplo.

Semáforo e Mutex: desvantagens

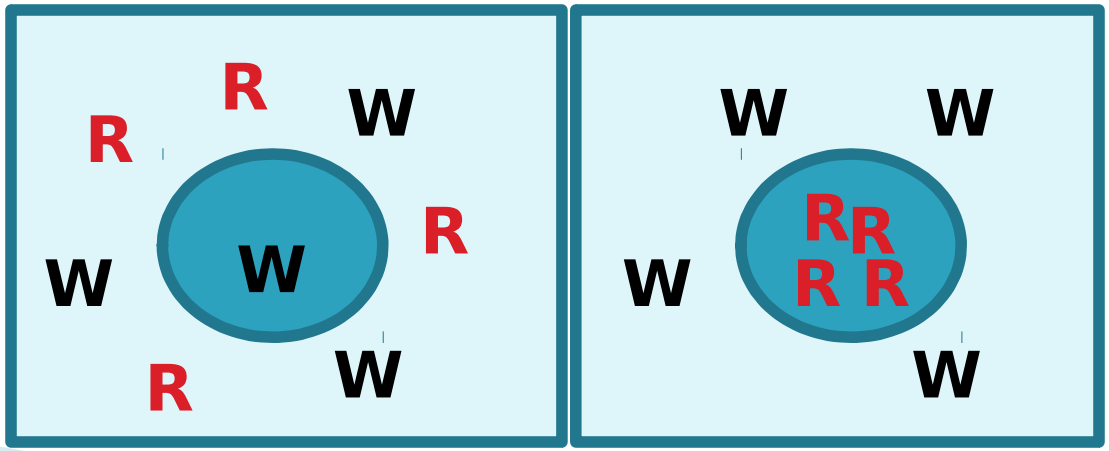
- Desvantagens do semáforo e do mutex:
  - São considerados mecanismos de difícil programação para sincronização (baixo nível).
  - Não são estruturados – é fácil cometer erros, por exemplo, esquecendo de realizar alguma operação P ou V (ou *lock* e *unlock*).
  - A detecção de erros é difícil.
  - São variáveis globais e podem existir vários deles ao mesmo tempo e aninhados ou entrelaçados.

Monitor

- É um mecanismo de abstração de dados.
  - Encapsula um conjunto de dados que só pode ser manipulado a partir de procedimentos que implementam as operações sobre estes dados.
- A exclusão mútua é oferecida de forma implícita.
  - Os procedimentos de um monitor não podem ser executados de forma concorrente.
- As variáveis de condição utilizadas em conjunto permitem a execução de sincronização condicional (de forma explícita).

Monitor: leitores-escritores (exemplo)

- Escritores necessitam acesso mutuamente exclusivo a um banco de dados.
- Leitores - como um grupo – também necessitam de acesso mutuamente exclusivo ao banco de dados com relação a qualquer escritor.



Monitor: leitores-escritores (exemplo)

```
monitor RW_controller{
  int nr=0, nw = 0; //(nr==0 || nw ==0) && nw <=1 (invariante)
  cond oktoread; //sinalizada quando nw==0
  cond oktowrite; //sinalizada quando nr==0 e nw==0

  procedure request_read(){
        while(nw>0) wait(oktoread);
        nr++;
  }
  procedure release_read(){
        nr--;
        if(nr==0) signal(oktowrite);
  }
  procedure request_write(){
        while(nr>0 || nw>0) wait(oktowrite);
        nw++;
  }
  procedure release_write(){
        nw--;
        signal(oktowrite);
        signal_all(oktoread);
  }
}
```

Fonte: Andrews, 2000

Threads em Java: recursos

- Recursos relacionados
  - synchronized: define um método ou bloco mutuamente exclusivo
  - wait: bloqueia a thread à espera de um sinal
    - Só pode ser invocado de dentro de um synchronized.
    - Libera o lock do synchronized.
    - Coloca a thead em uma fila.
  - notify: sinaliza uma thread bloqueada em wait.
  - notify_all: broadcast do sinal para todas as threads que estejam bloqueadas.

Threads em Java (exemplo)

```
//  Readers/Writers with concurrent read or exclusive write
//
// Usage:
//         javac rw.real.java
//         java Main rounds

class RWbasic {          // basic read or write
  protected int data = 0;  // the "database"
  protected void read() {
    System.out.println("read:  " + data);
  }
  protected void write() {
    data++;
    System.out.println("wrote:  " + data);
  }
}
```

Threads em Java (exemplo)

```
class ReadersWriters extends RWbasic {  // Readers/Writers
  int nr = 0;
  private synchronized void startRead() {
    nr++;
  }
  private synchronized void endRead() {
    nr--;
    if (nr==0) notify();  // awaken waiting Writers
  }
  public void read() {
    startRead();
    System.out.println("read:  " + data + "  nr= "+nr);
    endRead();
  }
  public synchronized void write() {
    while (nr>0)
      try { wait(); }
        catch (InterruptedException ex) {return;}
    data++;
    System.out.println("wrote:  " + data);
    notify();    // awaken another waiting Writer
  }
}
```

Threads em Java (exemplo)

```
class Reader extends Thread {
  int rounds;
  ReadersWriters RW;
  public Reader(int rounds, ReadersWriters RW) {
    this.rounds = rounds;
    this.RW = RW;
  }
  public void run() {
    for (int i = 0; i<rounds; i++) {
      RW.read();
    try{sleep(5);}catch (InterruptedException ex){return;}
    }
  }
}
```

Threads em Java (exemplo)

```
class Writer extends Thread {
  int rounds;
  ReadersWriters RW;
  public Writer(int rounds, ReadersWriters RW) {
    this.rounds = rounds;
    this.RW = RW;
  }
  public void run() {
    for (int i = 0; i<rounds; i++) {
      RW.write();
    try{sleep(10);}catch (InterruptedException ex)
{return;}
    }
  }
}
```

Threads em Java (exemplo)

```
class Main {  // driver program -- two readers and one writer
  static ReadersWriters RW = new ReadersWriters();
  public static void main(String[] arg) {
    int rounds = Integer.parseInt(arg[0],10);
    new Reader(rounds, RW).start();
    new Reader(rounds, RW).start();
    new Writer(rounds, RW).start();
  }
}
```

Bibliografia

CAVALHEIRO, G. G. H., COSTA, C. M., STRINGHINI, D.Programação Concorrente: Threads, MPI e PVM In: ERAD 2002 - 2a Escola Regional de Alto Desempenho ed.Porto Alegre : SBC, 2002, p. 31-65.

ANDREWS, G.R. Foundations of Multithreaded, Parallel and Distributed Programming. Reading: Addison-Wesley, 2000.