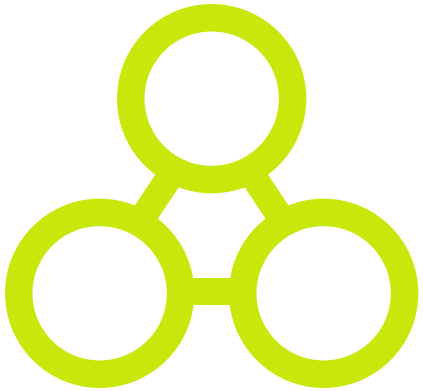


nemo

ontology & conceptual modeling research group

Linguagens de Programação
9 – Concorrência

Vítor E. Silva Souza

(vitor.souza@ufes.br)

http://www.inf.ufes.br/~vitorsouza

Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo

Conteúdo do curso

- Introdução;
- Amarrações;
- Valores e tipos de dados;
- Variáveis e constantes;
- Expressões e comandos;
- Modularização;
- Polimorfismo;
- Exceções;
- Concorrência;
- Avaliação de linguagens.

- Estes slides foram baseados em:
  - Slides do prof. Flávio M. Varejão;
  - Livro “Linguagens de Programação – Conceitos e Técnicas” (Varejão):
    - Este capítulo em particular tem contribuições de Jociel Cavalcante Andrade e Mariella Berger.

Conceituação

- Situação na qual diferentes processos competem pela utilização de algum recurso (processador, periféricos, etc.) ou cooperam para a realização de uma mesma tarefa;
- Primeiros sistemas eram mono-tarefa;
  - Alguns recursos são lentos (ex.: impressora) e o processador fica ocioso;
- Sistemas modernos usam *time slices*;
- Máquinas modernas possuem múltiplos processadores.

Motivação

- Aumento do número de computadores com múltiplos processadores e múltiplos núcleos;
  - Já se vendem hoje servidores com mais de uma centena de núcleos;
- O paralelismo de processos implementado pelo SO não é mais suficiente para ocupar todos os núcleos;
- Cabe ao programador, agora, paralelizar seu código.

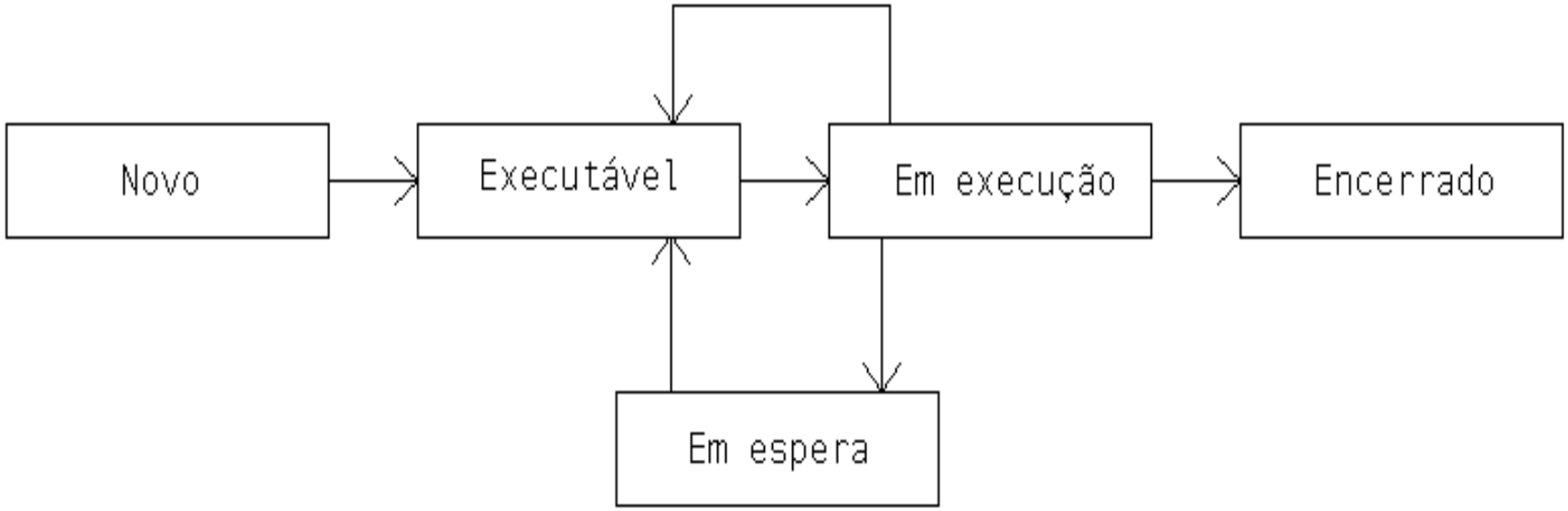
Processos

- Programas de computador são sequências de instruções passíveis de serem executadas por um processador;
- Quando estes programas estão sendo executados, eles são chamados de processos;
- Diferentemente dos programas, os processos são entidades ativas cujo estado é alterado durante a sua execução:
  - Múltiplos processos para o mesmo programa. Ex.: executar o programa zip para compactar dois conjuntos de arquivos diferentes.

Estado de um processo

- Durante a execução de um processo, o seu estado é modificado;
- O estado de um processo é definido pela atividade que está realizando;
- Possíveis estados de um processo:
  - Novo (sendo criado);
  - Executável (aguardando o processador);
  - Em execução (usando o processador);
  - Em espera (aguardando algum evento);
  - Encerrado (finalizado).

Transição de estados



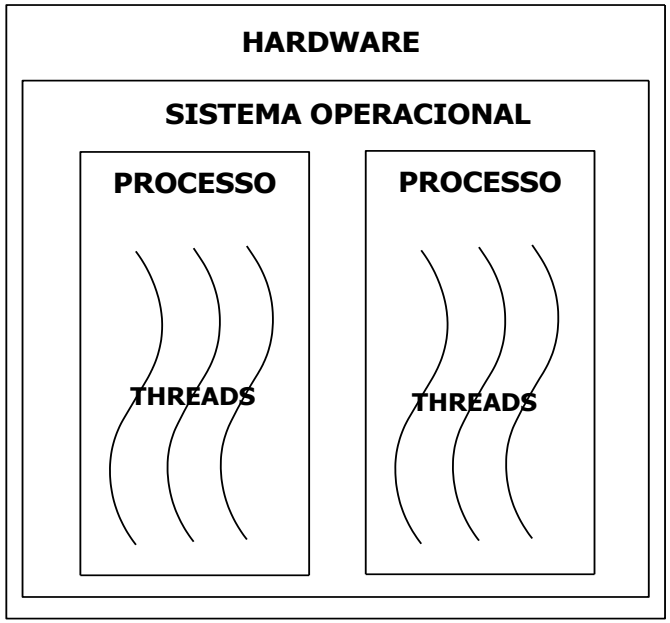
Sempre que um processo passa do estado em execução para o estado executável ou para o estado em espera, o sistema operacional salva o contexto de execução daquele processo.

Threads

- *Threads* são fluxos de execução concorrentes que compartilham recursos do processo do qual são originários;
  - Linhas/fluxos de execução;
  - “Processos leves”;
- A concorrência intra-processos agrava a concorrência inter-processos:
  - *Thread* compartilha recursos com outras *threads* do mesmo processo e com *threads* de outro processo.

Vantagens das *threads*

- São mais leves (troca de contexto mais rápida);
- Compartilham memória com o processo que o criou e com as demais *threads* (comunicação mais rápida);
- Possibilitam a utilização de mais de um método ou função de uma mesma aplicação, simultaneamente.



Processos concorrentes

- Os programas concorrentes são não-determinísticos;
- Isso acarreta alguns problemas:
  - Indeterminismo;
  - Lockout (trancamento);
  - Deadlock (impasse);
  - Starvation (inanição).

Indeterminismo da concorrência

- Não sabemos a ordem que as instruções *assembly* serão executadas;
- Problema com variáveis compartilhadas;
- No código abaixo, o conteúdo do endereço de memória i ([i]) é 100, e i é uma variável compartilhada:

Processo 1:

```
mov ax, [i]
mul ax, 2
mov [i], ax
```

i = i * 2

Processo 2:

```
mov ax, [i]
sub ax, 50
mov [i], ax
```

i = i - 50

Indeterminismo da concorrência

a. Valor final de *i*: *150*

Processo1	Processo2
mov ax, [i]	
mul ax, 2	
mov [i], ax	
	mov ax, [i]
	sub ax, 50
	mov [i], ax

c. Valor final de *i*: *200*

Processo1	Processo2
mov ax, [i]	
	mov ax, [i]
	sub ax, 50
	mov [i], ax
mul ax, 2	
mov [i], ax	

b. Valor final de *i*: *100*

Processo1	Processo2
	mov ax, [i]
	sub ax, 50
	mov [i], ax
mov ax, [i]	
mul ax, 2	
mov [i], ax	

d. Valor final de *i*: *50*

Processo1	Processo2
	mov ax, [i]
mov ax, [i]	
mul ax, 2	
mov [i], ax	
	sub ax, 50
	mov [i], ax

Outros problemas de concorrência

- Lockout (trancamento):
  - Vários processos aguardando um evento (recurso ocupado), reduzindo o desempenho;
- Deadlock (impasse):
  - Dois (ou mais) processos aguardando evento que só pode ser produzido pelo outro;
- Starvation (inanição):
  - Processo com baixa prioridade nunca obtém um recurso (outros “passam na frente”).

Tipos de interação

- Sem interferência (competição);
- Com interferência (cooperação).

Tipos de interação: competição

- Processos independentes que apenas concorrem com outros processos pela utilização de um mesmo recurso;
- Sincronização para utilização dos recursos é feita pelo sistema operacional, não provocando qualquer dificuldade para os programadores;
- Não há compartilhamento de dados, portanto a execução é efetivamente determinística.

Tipos de interação: cooperação

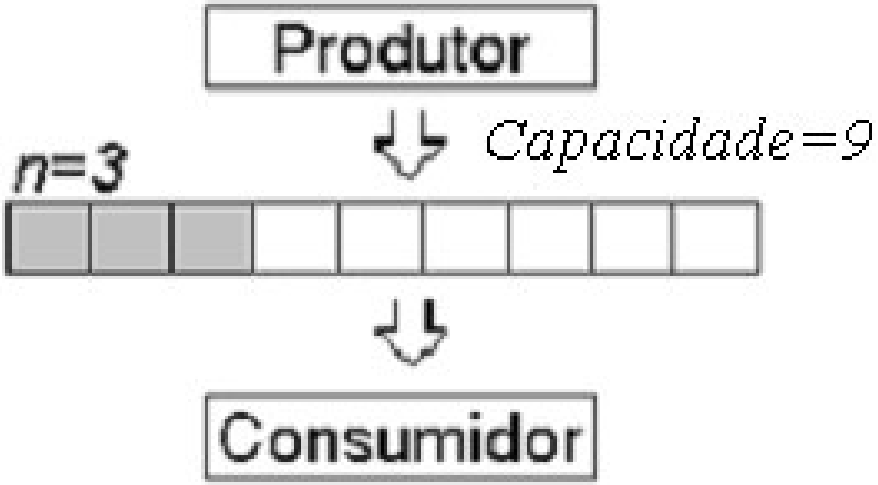
- Processo afeta ou é afetado pela execução de outro processo em prol da realização de uma atividade;
- Deve ser gerenciado pelo programador/LP;
- Para que ocorra a cooperação entre processos é necessário que exista uma forma de comunicação entre eles:
  - Troca de mensagens: obrigatório para processos distribuídos, gera overhead;
  - Compartilhamento de memória: mais eficientes, pode causar inconsistência, exige sincronia.

Troca de mensagens

- Duas chamadas de sistema básicas:
  - *send* (envio);
  - *receive* (recepção);
- Dois tipos de comunicação por troca de mensagens:
  - Direta (processo-processo);
  - Indireta (caixa postal, suporta múltiplos processos).
- Comunicação de processos através de troca de mensagens:
  - Bloqueante (síncrono);
  - Não-bloqueante (assíncrono);
- *Rendezvous* (encontro): a comunicação só ocorre quando o remetente e o destinatário se encontram.

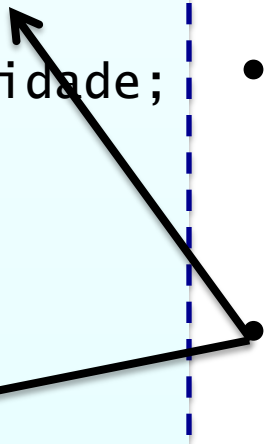
Compartilhamento de memória

- Necessidade de proteção de acesso a dados compartilhados para evitar inconsistências;
- Ex.: no problema do Produtor e do Consumidor há um processo que produz algo e um processo que utiliza o que foi produzido.



Produtor e consumidor

```
// Inicialização (de ambos).
fim = 0;
ini = 0;
n = 0;

// Código do produtor.
for (i = 0; i<1000; i++) {
  while (n == capacidade);
  buf[fim] = produzir(i);
  fim = (fim + 1) % capacidade;
  n++;
}

// Código do consumidor.
for (i=0; i<1000; i++) {
  while (n == 0);
  consumir(buf[ini]);
  ini = (ini + 1) % capacidade;
  n--;
}
```



- Início da execução do consumidor é dependente do término da execução do produtor;
- A utilização de processos concorrentes para resolver esse problema é mais apropriada;
- Deve haver algum tipo de sincronização de modo a garantir acesso exclusivo a sua **região crítica** (a var. n);
- Uso do "*busy wait*" (ocupa o processador inutilmente);
- (De)Incremento traduzido para múltiplas operações assembly, pode gerar indeterminismo.

Sincronização

Semáforos

Programação Concorrente Estruturada

Semáforos (Dijkstra)

- Um semáforo é um tipo abstrato de dados que possui um valor inteiro, uma lista de processos em espera e duas operações:
  - P (do holandês *Proberen*, testar);
  - V (do holandês *Verhogen*, incrementar);
- A implementação do semáforo deve ser atômica:
  - Garantido por LPs que oferecem o recurso.

Semáforos

- Para garantir exclusão mútua a uma determinada região (região crítica), cada processo deve:
  - Chamar a operação P antes de acessar tal região; e
  - Chamar a operação V após sair dessa região;
- Operação P sobre um semáforo S:
  - Testa se o processo que executou P(S) pode ou não entrar na região crítica;
- Operação V sobre um semáforo S:
  - Sinaliza ao semáforo que o processo não está mais na região crítica e retira outro processo da fila de espera.

Semáforos

```
// S.valor começa com 1.

P(S)
  S.valor -= 1;
  Se (S.valor < 0)
       // Bloqueia o processo e insere em S.fila.

V(S)
  S.valor += 1;
  Se (S.valor <= 0)
     // Retira algum processo de S.fila e o
     // coloca em execucao.
```

O algoritmo acima pode ser facilmente generalizável para semáforos que possuam N recursos compartilhados.

Produtor e consumidor com semáforos

```
// Inicialização (ambos).
fim = 0;
ini = 0;
n = 0;
semaforo S;
S.valor = 1;

// Código do produtor:
for (i=0; i<1000; i++) {
  while (n == capacidade);
  buf[fim] = produzir(i);
  fim = (fim + 1) %
                capacidade;
  P(S);
  n++;
  V(S);
}
```

```
// Código do consumidor:
for (i=0; i<1000; i++) {
  while (n == 0);
  consumir(buf[ini]);
  ini = (ini + 1) %
                capacidade;
  P(S);
  n--;
  V(S);
}

// Obs.:
// Produtor e consumidor
// executam em paralelo.
```

Semáforos – desvantagens

- LP garante P(S) e V(S) atômicas, porém o programador deve saber usá-las:
  - Se fizer V(S) antes e P(S) depois, há possível violação de acesso exclusivo;
  - Se há P(S) mas não V(S), pode bloquear outros processos infinitamente;
  - Se há V(S) mas não P(S) há também violação de acesso exclusivo.

Programação Concorrente Estruturada

- Abstraem os conceitos de semáforo para lidar com os problemas anteriores;
- Mecanismos deixam para o compilador a responsabilidade de garantir o acesso exclusivo à região crítica;
  - Regiões críticas condicionais;
  - Monitores.

Regiões críticas condicionais

- Variáveis compartilhadas são declaradas como tal;
- Regiões críticas são demarcadas com uma palavra-chave específica;
- O compilador insere `P(S)` e `V(S)` automaticamente.

Monitores

- Mecanismos de sincronização compostos por um conjunto de variáveis, procedimentos e estruturas de dados dentro de um módulo;
- Finalidade é a implementação automática da sincronização, garantindo exclusão mútua entre seus procedimentos.

Monitores

- Exemplo:
  - Subprogramas `p1()` e `p2()` são definidos em um monitor;
  - Possuem acesso exclusivo.

```
Monitor <nome-do-monitor> {
  // Declarações de variáveis.

  p1(...) {
     ...
  }

  p2(...) {
     ...
  }
}
```

Ausência de mecanismos em LPs

- Algumas LPs, tal como C, não oferecem mecanismos próprios para lidar com concorrência;
- É possível criar e manipular processos e threads e lidar com problemas de concorrência através de:
  - Chamadas de sistema;
  - Bibliotecas de funções específicas da plataforma de execução.

Concorrência em C: chamadas fork

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void imprime(char *arg) {
  int i;
  for (i = 0; i < 1000000; i++)    // 1 milhão.
    printf(arg);
  exit(0);
}

int main() {
  int pid;
  int estado_filho;
  pid = fork();                              // Continua...
```

Concorrência em C: chamadas fork

```
if (pid < 0) { // Erro!
   perror("Erro de criação do primeiro filho");
}
else {
   if (pid == 0) { // É um processo filho.
      imprime("-");
   }
}

pid = fork();
if (pid < 0) { // Erro!
   perror("Erro de criação do segundo filho.");
} else {
   if (pid == 0) {// É um processo filho.
      imprime(".");
   }
}
                                        // Continua...
```

Concorrência em C: chamadas fork

```
  wait(&estado_filho); // Aguarda o término de um filho.
  wait(&estado_filho); // Aguarda o outro filho.
  return 0;
}
```

Sincronização de processos em C

- Pode ser implementada com semáforos usando chamadas de sistema;
- No entanto, não é uma tarefa fácil;
- Dependente de SO, portanto não nos aprofundaremos no assunto…

Concorrência em C: *POSIX Threads*

- POSIX = *Portable Operating System Interface for Unix*;
  - Define como programas Unix devem se comunicar, visando portabilidade;
- O *POSIX threads* é um padrão adotado para manipular *threads* em C;
- A API padrão do *pthreads* conta com aproximadamente 60 funções.

Concorrência em C: *pthreads*

```c
#include <stdio.h>
#include <string.h>
#include <pthread.h>

void *imprime(void *arg) {
  int i;
  for (i=0; i<1000000; i++)
     printf((char *)arg);
  pthread_exit(0);
}

int main() {
  pthread_t tid1, tid2;
  pthread_attr_t attr;
  char msg1[30];
  char msg2[30];
  strcpy(msg1, ".");
  strcpy(msg2, "-");
  pthread_attr_init(&attr);

  pthread_create (&tid1,
     &attr, imprime, msg1);
  pthread_create(&tid2,
     &attr, imprime, msg2);

  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  return 0;
}
```

Para compilar, usa-se a flag -lpthread

Semáforos com *pthreads*

- Mecanismo de exclusão mútua (*mutex*) com operações `lock` (`P(S)`) e `unlock` (`V(S)`)
- Não é necessário efetuar *busy wait*: pode-se colocar uma *thread* em espera pelo mecanismo de variáveis condicionais;
- O código a seguir simula o mecanismo de monitores em C...

Produtor e consumidor com *pthreads*

```
#include<stdio.h>
#include<pthread.h>
#include <stdlib.h>

#define CAPACIDADE 10

typedef struct BufferLim {
  int buf[CAPACIDADE];
  int fim;
  int ini;
  int n;
  pthread_mutex_t mut;
  pthread_cond_t vazio;
  pthread_cond_t cheio;
} BufferLimitado;
```

Produtor e consumidor com *pthreads*

```
void *produtor(void *arg) {
  BufferLimitado *bl = (BufferLimitado *)arg;
  int i;

  for (i = 0; i < 1000; i++) {
    pthread_mutex_lock(&bl->mut);
    while (bl->n == CAPACIDADE)
      pthread_cond_wait(&bl->cheio, &bl->mut);

    bl->buf[bl->fim] = i;
    printf("produzido: %d\n", bl->buf[bl->fim]);
    bl->fim = (bl->fim + 1) % CAPACIDADE;
    bl->n++;
    pthread_mutex_unlock(&bl->mut);
    pthread_cond_signal(&bl->vazio);
  }
  pthread_exit(0);
}
```

Produtor e consumidor com *pthreads*

```
void *consumidor(void *arg) {
  BufferLimitado *bl = (BufferLimitado *)arg;
  int i;

  for (i = 0; i < 1000; i++) {
    pthread_mutex_lock(&bl->mut);
    while (bl->n == 0)
      pthread_cond_wait(&bl->vazio, &bl->mut);

    printf("consumido: %d\n”, bl->buf[bl->ini]);
    bl->ini = (bl->ini + 1) % CAPACIDADE;
    bl->n--;
    pthread_mutex_unlock(&bl->mut);
    pthread_cond_signal(&bl->cheio);
   }

   pthread_exit(0);
}
```

Produtor e consumidor com *pthreads*

```
int main() {
  BufferLimitado bl;
  pthread_t tid1, tid2;
  bl.fim = bl.ini = bl.n = 0;
  pthread_mutex_init(&bl.mut, NULL);
  pthread_cond_init(&bl.cheio, NULL);
  pthread_cond_init(&bl.vazio, NULL);
  pthread_create(&tid1, NULL, produtor, &bl);
  pthread_create(&tid2, NULL, consumidor, &bl);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);
  return 0;
}
```

Threads em Java

- Java possui amplo suporte à programação concorrente;
  - Suporte básico com `java.lang.Thread`;
  - API de alto nível em `java.util.concurrent`.
- Dois modos de se criar *threads* em Java:
  - Herdando da classe `Thread`: mais simples;
  - Implementando a interface `Runnable`: mais flexível.

A abordagem com Runnable é preferível.

Threads em Java: herdando de Thread

```
class Carro extends Thread {
  public Carro(String nome) { super(nome); }

  public void run() {
     for (int i = 0; i < 10; i++) {
        try {
           sleep((int)(Math.random()*1000));
        }
        catch (Exception e) { e.printStackTrace(); };
        System.out.print(getName());
        for (int j = 0; j < i; j++)
           System.out.print("--");
        System.out.println(">");
     }
        System.out.println(getName() +
                                   " completou a prova.");
  }
}
```

Threads em Java: herdando de Thread

```
public class Corrida {
  public static void main(String[] args) {
     Carro carroA = new Carro("Barrichelo");
     Carro carroB = new Carro("Schumacher");
     carroA.start();
     carroB.start();

     try {
        carroA.join();
     } catch (Exception e) { e.printStackTrace(); }

     try {
        carroB.join();
     } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Threads em Java: herdando de Thread

```
Barrichelo>
Schumacher>
Schumacher-->
Barrichelo-->
Schumacher---->
Barrichelo---->
Schumacher------>
Barrichelo------>
Barrichelo-------->
Schumacher-------->
Barrichelo---------->
Schumacher---------->
Barrichelo------------>
Schumacher------------>
Barrichelo-------------->
Schumacher-------------->
Barrichelo---------------->
Barrichelo------------------>
Barrichelo completou a prova.
Schumacher---------------->
Schumacher------------------>
Schumacher completou a prova.
```

Possível resultado da execução.

Threads em Java: implement. Runnable

```
class Carro2 implements Runnable {
  private String nome;
  public Carro2(String nome) { this.nome = nome; }

  public void run() {
     for (int i = 0; i < 10; i++) {
        try {
           Thread.sleep((int)(Math.random()*1000));
        }
        catch (Exception e) { e.printStackTrace(); };
        System.out.print(nome);
        for (int j = 0; j < i; j++)
           System.out.print("--");
        System.out.println(">");
     }
        System.out.println(nome + " completou a prova.");
  }
}
```

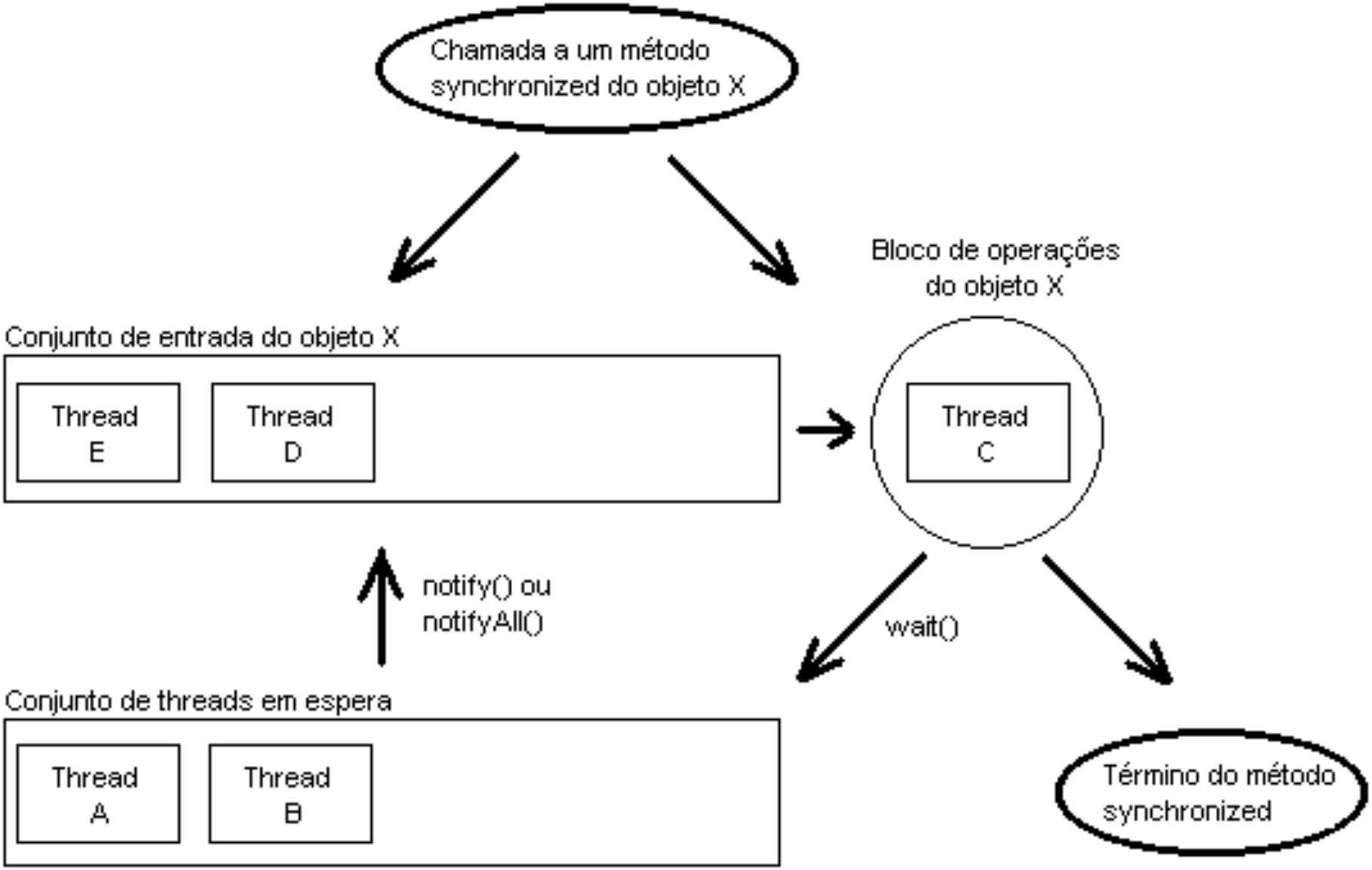
Threads em Java: implement. Runnable

```
public class Corrida2 {
  public static void main(String[] args) {
     Carro2 carroA = new Carro2("Barrichelo");
     Carro2 carroB = new Carro2("Schumacher");
     Thread threadA = new Thread(carroA);
     Thread threadB = new Thread(carroB);
     threadA.start();
     threadB.start();

     try {
        threadA.join();
     } catch (Exception e) { e.printStackTrace(); }

     try {
        threadB.join();
     } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Métodos sincronizados em Java

- Todo objeto em Java possui um bloco de operações único;
- Quando métodos de um objeto em Java são declarados como `synchronized`:
  - A JVM faz com que *threads* chamem esses métodos em exclusão mútua;
  - Se uma *thread* chama um método `synchronized`, verifica se há outra de posse do bloco de operações;
  - Se há, aguarda na fila. Se não, toma posse do bloco.

Métodos sincronizados em Java



P&C com synchronized em Java

```
class BufferLimitado {
  private int capacidade;
  private int n;
  private int[] buffer;
  private int fim;
  private int ini;

  public BufferLimitado(int capacidade) {
     this.capacidade = capacidade;
     n = 0;
     fim = 0;
     ini = 0;
     buffer = new int[capacidade];
  }

  // Continua...
```

P&C com synchronized em Java

```
  public synchronized void inserir(int elemento) throws
                                        InterruptedException {
     while (n == capacidade) wait();
     buffer[fim] = elemento;
     fim = (fim + 1) % capacidade;
     n = n + 1;
     notify();
  }
  public synchronized int retirar() throws
                                        InterruptedException {
     while (n == 0) wait();
     int elem = buffer[ini];
     ini = (ini + 1) % capacidade;
     n = n - 1;
     notify();
     return elem;
  }
}
```

P&C com synchronized em Java

```
class Produtor extends Thread {
  private BufferLimitado buffer;
  public Produtor(BufferLimitado buffer) {
     this.buffer = buffer;
  }
  public void run() {
     int elem;
     while (true) {
        elem = (int)(Math.random() * 10000);
        try {
           buffer.inserir(elem);
           System.out.println("produzido: " + elem);
           Thread.sleep((int)(Math.random() * 1000));
        } catch (InterruptedException e) {
           System.out.println(Erro Produtor: " + e);
        }
     }
  }
}
```

P&C com synchronized em Java

```
class Consumidor extends Thread {
  private BufferLimitado buffer;
  public Consumidor(BufferLimitado buffer) {
     this.buffer = buffer;
  }
  public void run() {
     int elem;
     while (true) {
        try {
           elem = buffer.retirar();
           System.out.println("consumido: " + elem);
           Thread.sleep((int)(Math.random() * 1000));
        } catch (InterruptedException e) {
           System.out.println("Erro Consumidor: " + e);
        }
     }
  }
}
```

P&C com synchronized em Java

```
public class Fabrica {
  public static void main(String args[]) throws
                                        InterruptedException {

    BufferLimitado buffer = new BufferLimitado(10);
    Produtor produtor = new Produtor(buffer);
    Consumidor consumidor = new Consumidor(buffer);

    produtor.start();
    consumidor.start();

    produtor.join();
    consumidor.join();
  }
}
```

Thread-safe

- Uma classe é *thread-safe* quando pode ser usada por N *threads*. Isso significa:
  - Que ela foi preparada para tal com métodos `synchronized`, etc.;
  - Que ela tem performance inferior a uma classe equivalente "*thread-unsafe*";
  - Ex.: `BlockingQueue`, `ConcurrentMap`, `Vector` (`java.util.concurrent` / `java.util`).

Estratégias de paralelismo

- Filas de trabalho + grupos de *threads* (*work queues + thread pools*):
  - Classes *thread-safe*, uso básico de Thread;
  - Fundamental no AWT/Swing (EDT, SwingWorker);
- *Fork-Join*: dividir para conquistar;
  - Novidade no Java 7;
- *Select / map / reduce*: divide grande quantidade de dados em lotes para processamento paralelo;
  - Lisp, Smalltalk;
  - *Bulk data operations*, no Java 8.

Fork-Join em Java

```
public class TesteForkJoin {
  private static final int QTD = 100_000_000;
  private static double[] gerarNumeros(int qtd) {
     double[] numeros = new double[qtd];
     Random rand = new Random(System.currentTimeMillis());
     for (int i = 0; i < qtd; i++)
        numeros[i] = rand.nextDouble() * 10d;
     return numeros;
  }
  public static void main(String[] args) throws Exception {
     double[] numeros = gerarNumeros(QTD);
     long inicio = System.nanoTime();
     double soma = SomadorParalelo.somarParalelo(numeros);
     // ou: SomadorSequencial.somarSequencialmente(numeros);
     long fim = System.nanoTime();
     double tempo = (fim - inicio) / 1_000_000.0;
     System.out.println("Média = " + (soma / QTD));
     System.out.println("Tempo = " + tempo + "ms");
  }
}
```

Fork-Join em Java

```
public class SomadorSequencial {
  public static double somarSequencialmente(double[] numeros)
{
    double resultado = 0d;
    for (int i = 0; i < numeros.length; i++)
      resultado += numeros[i];
    return resultado;
  }
}
```

```
import java.util.concurrent.*;

public class SomadorParalelo {
  public static double somarParalelo(double[] numeros) {
    TarefaSoma tarefa =
      new TarefaSoma(new VetorNumeros(numeros));
    ForkJoinPool pool = new ForkJoinPool();
    pool.invoke(tarefa);
    return tarefa.resultado;
  }
}
```

Fork-Join em Java

```
class TarefaSoma extends RecursiveAction {
  private static final int LIMITE_SEQUENCIAL = 2_000_000;
  final VetorNumeros vetor;
  double resultado;
  TarefaSoma(VetorNumeros vetor) {
     this.vetor = vetor;
  }
  protected void compute() {
     if (vetor.tamanho < LIMITE_SEQUENCIAL)
        resultado = vetor.somar();
     else {
        TarefaSoma esquerda
           = new TarefaSoma(vetor.subVetorEsquerda());
        TarefaSoma direita
           = new TarefaSoma(vetor.subVetorDireita());
        invokeAll(esquerda, direita);
        resultado = esquerda.resultado + direita.resultado;
     }
  }
}
```

Objeto que aponta para áreas diferentes do mesmo vetor.

Fork-Join em Java

- Solução Java é:
  - Genérica: tarefa específica a cargo do programador;
  - Portável: independente do hardware;
- Ressalvas:
  - Indicado apenas para tarefas que fazem uso intenso da CPU, sem muitas operações de I/O;
  - Há um *overhead* de gerenciamento do paralelismo (divisão do vetor, combinação dos resultados, gerência das *threads*), portanto é necessário testar para quais valores vale a pena paralelizar.

Com 2 núcleos, somente a partir de ~ 100 milhões de números o somador paralelo foi mais rápido.

Módulos concorrentes

- A execução de programas na linguagem de programação Ada consiste na execução de uma ou mais tarefas (*tasks*);
- ADA disponibiliza dois mecanismos de comunicação entre tarefas:
  - Passagem de mensagens (*rendevouz*);
  - Objetos protegidos.

Passagem de mensagem em Ada

- Feita quando uma tarefa chama uma entrada de outra tarefa;
- Ocorre quando os dois lados, cliente e servidor, se encontram;
- Uma tarefa possui uma especificação e um corpo:
  - Na especificação podem-se declarar *entry points* (pontos de entrada);
  - No corpo é feita a implementação do restante da descrição da tarefa;
- Os pontos de entrada são mutuamente exclusivos.

Passagem de mensagem em Ada

```
-- Esquema de uma tarefa:
task <nome-da-task> is
  ...
  entry <nome-da-entrada>
  ...
end <nome-da-task>

task body <nome-da-task> is
  ...
  accept <nome-da-entrada>
  do
    <comandos>
  end <nome-da-entrada>
  ...
end <nome-da-task>
```

Passagem de mensagem em Ada

```
-- Implementação de uma tarefa:
with Text_IO; use Text_IO;
procedure teste is
  task type Carro is
    entry iniciar(id: integer);
    entry andar_para_frente;
    entry andar_para_tras;
  end Carro;
  task body Carro is
    posicao: integer;
    meu_id: integer;
  begin
    accept iniciar(id: integer) do
      posicao := 0;
      meu_id := id;
    end iniciar;
    Put_Line("O carro " & Integer'Image(meu_id) &
      " está na posição " & Integer'Image(posicao));
```

Passagem de mensagem em Ada

```
  select
     accept andar_para_frente do
        posicao := posicao + 1;
     end andar_para_frente;
  or
     accept andar_para_tras do
        posicao := posicao - 1;
     end andar_para_tras;
  end select;
  Put_Line("O carro " & Integer'Image(meu_id) &
     " está na posição " &  Integer'Image(posicao));
  end Carro;
  carro1, carro2 : Carro;
begin
  carro1.iniciar(1); carro2.iniciar(2);
  carro1.andar_para_frente;
  carro2.andar_para_tras;
end teste;
```

Passagem de mensagem em Ada

- Possível resultado:
  - O carro 1 esta na posicao 0
  - O carro 1 esta na posicao 1
  - O carro 2 esta na posicao 0
  - O carro 2 esta na posicao –1

Objetos protegidos em Ada

- Operações protegidas podem ser:
  - Procedimentos (*procedures*);
  - Funções (*functions*);
  - Entradas (*entries*);
- Chamadas a procedimentos e entradas protegidos são executadas em exclusão mútua:
  - As funções podem ser executadas em paralelo, mas não quando um procedimento ou entrada protegidos estão sendo executados.

Objetos protegidos em Ada

```
-- Esquema de uma unidade protegida simples:
protected type <nome-da-unidade> is
  function <nome-da-função> return <tipo-retorno>;
  procedure <nome-do-procedimento>;
  entry <nome-da-entrada>;
  <declaração de variáveis>
end <nome-da-unidade>;

-- Continua...
```

Objetos protegidos em Ada

```
protected body <nome-da-unidade> is
  entry <nome-da-entrada> when <condição> is
  begin
    <comandos>
  end <nome-da-entrada>;

  procedure <nome-do-procedimento> is
  begin
    <comandos>
  end <nome-do-procedimento>;

  function <nome-da-função> return <tipo-retorno> is
  begin
    <comandos>
    return <valor>;
  end <nome-da-função>;
end <nome-da-unidade>;
```

Objetos protegidos em Ada

```
-- Implementação de um objeto protegido:
protected type Objeto_Sinal is
  entry Espera;
  procedure Sinal;
  function Esta_Aberto return boolean;

  private
     aberto: boolean := false;
end Objeto_Sinal;

-- Continua...
```

Objetos protegidos em Ada

```
protected body Objeto_Sinal is
  entry Espera when aberto is
  begin
    aberto := false;
  end Espera;

  procedure Sinal is
  begin
    aberto := true;
  end Sinal;

  function Esta_Aberto return boolean is
  begin
    return aberto;
  end Esta_Aberto;
end Objeto_Sinal;
```

P&C com objetos protegidos em Ada

```
package Fabrica is
  type Buffers is array(positive range <>) of integer;
  protected type BufferLimitado(capacid : natural) is
     entry Retirar(elem : out Integer);
     entry Inserir(elem : in Integer);
     private
        buf: Buffers(1..capacid);
        n: natural := 0;
        ini, fim: positive := 1;
        capacidade: natural := capacid;
  end BufferLimitado;
  type ABuffer is access bufferLimitado;
  task type Produtor is
     entry Iniciar(buf : in ABuffer);
  end Produtor;
  task type Consumidor is
     entry Iniciar(buf : in ABuffer);
  end Consumidor;
end Fabrica;
```

P&C com objetos protegidos em Ada

```
with Ada.Text_Io; use Ada.Text_Io;
package body Fabrica is
  protected body BufferLimitado is
    entry Retirar(elem : out integer) when n > 0 is
    begin
      elem := buf(ini);
      if (ini = capacidade) then ini := 1;
      else ini := ini + 1; end if;
      n := n - 1;
    end Retirar;

    entry Inserir(e : in integer) when n < capacidade is
    begin
      buf(fim) := e;
      if (fim = capacidade) then fim := 1;
      else fim := fim + 1; end if;
      n := n + 1;
    end Inserir;
  end BufferLimitado;
```

P&C com objetos protegidos em Ada

```
task body Produtor is
   elem: integer;
   pbuf: ABuffer;
   i: integer;

begin
   accept Iniciar(buf : in ABuffer) do
      pbuf := buf;
   end;

   for i in 0..1000 loop
      elem := i;
      pbuf.inserir(elem);
      Put_Line("Produzido: " & Integer'Image(elem));
   end loop;
end Produtor;
```

P&C com objetos protegidos em Ada

```
  task body Consumidor is
     elem: integer;
     cbuf: ABuffer;
     i: natural;

  begin
     accept Iniciar(buf : in aBuffer) do
        cbuf := buf;
     end;

     for i in 0..1000 loop
        cbuf.retirar(elem);
        Put_Line("Consumido: " & Integer'Image(elem));
     end loop;
  end Consumidor;
end Fabrica;
```

P&C com objetos protegidos em Ada

```
with Fabrica; use Fabrica;
with Ada.Text_Io; use Ada.Text_Io;

procedure Teste is
  prod: Produtor;
  cons: Consumidor;
  buf: ABuffer := new BufferLimitado(10);

begin
  prod.Iniciar(buf);
  cons.Iniciar(buf);
end Teste;
```

Conclusões

- Concorrência: benefícios e desafios;
- Mecanismos como semáforos e monitores foram propostos para ajudar;
- Algumas LPs dão suporte. Exemplos:
  - C deixa a tarefa com o programador;
  - Java oferece a classe `Thread` e o `synchronized`, API de alto nível `java.util.concurrent`;
  - Ada oferece troca de mensagens e objetos protegidos.

http://nemo.inf.ufes.br/