

# Linguagens de Programação

## Conceitos e Técnicas



Concorrência

# Conceituação

- Concorrência:
  - 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;
- Sistemas modernos usam time-slices;
- Máquinas modernas possuem múltiplos processadores.

# Processos

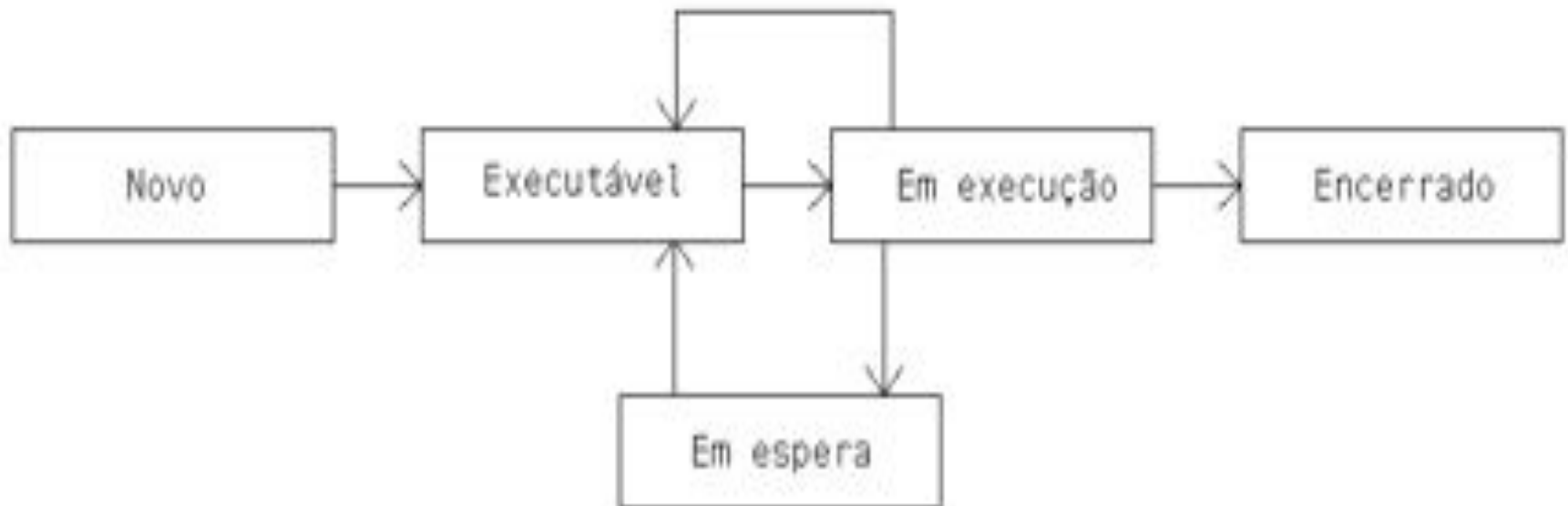


- Programas de computador são seqüências de instruções passíveis de serem executadas por um processador
- Quando estes programas estão sendo executados, eles são chamados de processos
- Diferentemente dos programas, os processos são entidades ativas cujo estado é alterado durante a sua execução

# 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

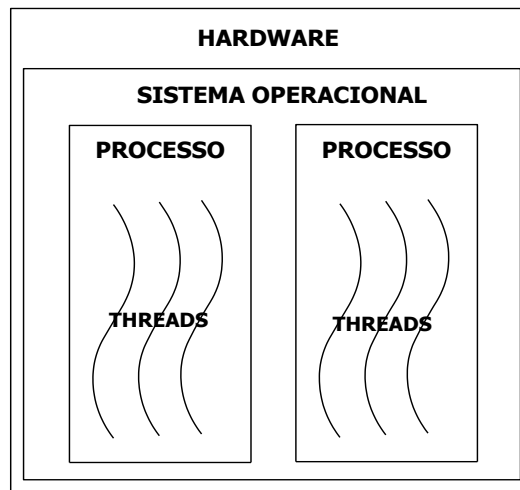


# Threads

- Threads são fluxos de execução concorrentes que compartilham recursos do processo do qual são originários
  - Linhas de execução;
  - "Processos leves"
- A concorrência intraprocessos agrava a concorrência interprocessos
  - Thread compartilha recursos com outras threads do mesmo processo e com threads de outro processo.

# Vantagens de Threads

- São mais leves
- Compartilham memória com o processo que o criou e com os demais threads
- 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
  - Lockout (trancamento)
  - Deadlock (impasse)
  - Starvation (inanição)
  - Indeterminismo



# Processos Concorrentes

## ■ Indeterminismo

- O exemplo leva em conta que 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

; Processo 2:

mov ax,  $[i]$

sub ax, 50

mov  $[i]$ , ax

# Processos Concorrentes

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

a. Valor final de *i*: 150

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

b. Valor final de *i*: 100

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

c. Valor final de *i*: 200

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

d. Valor final de *i*: 50

# Processos Concorrentes

- 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

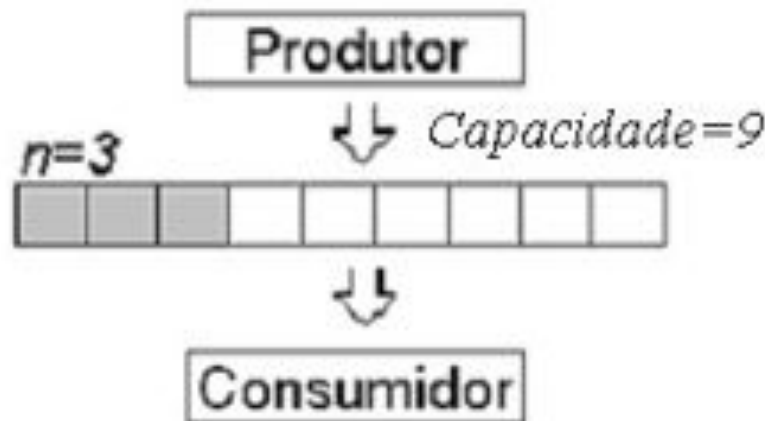
- Processo afeta ou é afetado pela execução de outro processo em prol da realização de uma atividade
- Deve ser gerenciado pelo programador/LP
- Para que ocorra a cooperação entre processos é necessário que exista uma forma de comunicação entre eles
  - Troca de mensagens
  - Compartilhamento de memória

# Troca de Mensagens

- Duas chamadas de sistema básicas
  - send (envio)
  - receive (recepção)
- Dois tipos de comunicação por troca de mensagens
  - Direta
  - Indireta.
- A 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
- 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

```
// inicializacao
fim = 0;
ini = 0;
n = 0;
// codigo do produtor
for (i=0; i<1000; i++) {
    while (n == capacidade);
    buf[fim] = produzir(i);
    fim = (fim + 1) % capacidade;
    n++;
}
// codigo do consumidor
for (i=0; i<1000; i++) {
    while (n == 0);
    consumir(buf[ini]);
    ini = (ini + 1) % capacidade;
    n--;
}
```

- Essa solução torna o início da execução do consumidor dependente do término da execução do produtor. A utilização de processos concorrentes para resolver esse problema é mais apropriada
- Para evitar o problema de indeterminismo deve haver algum tipo de sincronização entre os processos de modo a garantir acesso exclusivo a sua região crítica (a var. *n*)
- Uso do "busy wait" (ocupa o processador inutilmente).

# Sincronização



- Semáforos
- Programação concorrente estruturada

# Semáforos

- 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

# Produtor e Consumidor com Semáforos

```
// inicializacao
fim = 0;
ini = 0;
n = 0;
semaforo S;
S.valor = 1;

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

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

# 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

```
Monitor <nome-do-monitor> {  
    // declarações de variável  
    p1(...) {  
        ...  
    }  
    p2(...) {  
        ...  
    }  
}
```

- No exemplo é mostrada a definição de dois procedimentos p1 e p2
  - Eles possuem acesso exclusivo

# 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

# 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<10000000; i++)
        printf(arg);
    exit(0);
}
```

# Chamadas fork

```
int main() {  
    int pid;  
    int estado_filho;  
    pid = fork();  
    if (pid < 0) { // erro  
        perror("Erro de criação do primeiro filho");  
    } else {  
        if (pid == 0) { // filho  
            imprime("-");  
        }  
    }  
}
```

# Chamadas fork

```
pid = fork();
if (pid < 0) { // erro
    perror("Erro de criação do segundo filho.");
} else {
    if (pid == 0) { // filho
        imprime(".");
    }
}
wait(&estado_filho); // aguarda o término de um filho
wait(&estado_filho); // aguarda o término do 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.



# 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

# Pthreads

```
#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;
}
```

# 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, 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;  
}
```

# Classes Threads



- Dois modos de se criar threads em Java
  - Herdando da classe Thread
  - Implementando a interface Runnable



# Herdando da Classe 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) {};  
            System.out.print(getName());  
            for (int j=0; j<i; j++)  
                System.out.print("--");  
            System.out.println(">");  
        }  
        System.out.println(getName() + " completou a prova.");  
    }  
}
```

# Herdando da Classe 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) {}  
        try {  
            carroB.join();  
        } catch (Exception e) {}  
    }  
}
```

\_\_\_\_\_

\_\_\_\_\_

# Implementando 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) {};  
            System.out.print(nome);  
            for (int j=0; j<i; j++)  
                System.out.print("--");  
            System.out.println(">");  
        }  
        System.out.println(nome + " completou a prova.");  
    }  
}
```

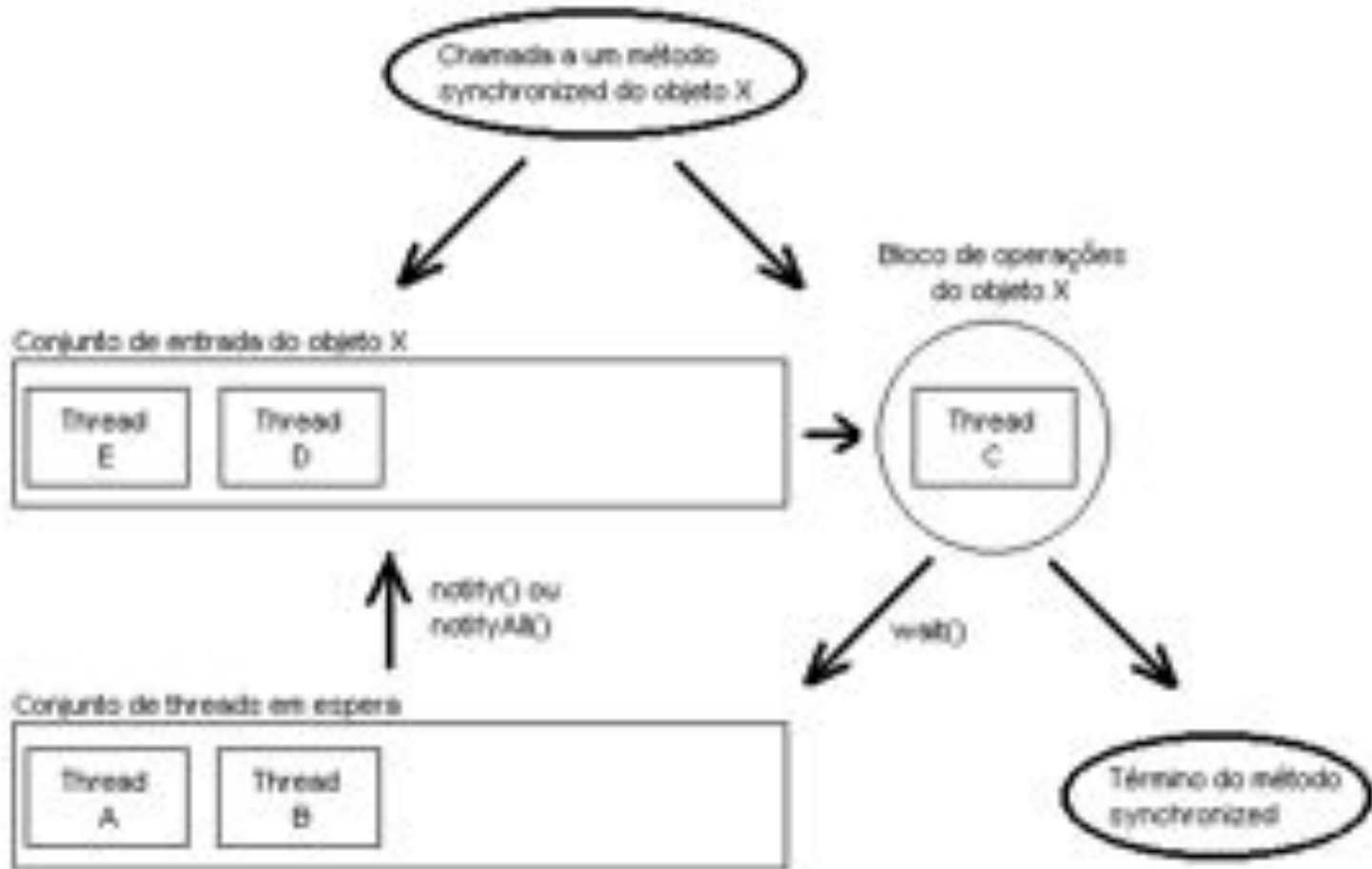
# Implementando Runnable

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

# Métodos Sincronizados

- 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



# Produtor e Consumidor com Métodos Sincronizados

```
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];  
    }
```



# Produtor e Consumidor com Métodos Sincronizados

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

# Produtor e Consumidor com Métodos Sincronizados

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

# Produtor e Consumidor com Métodos Sincronizados

```
class Produtor extends Thread {  
    private BufferLimitado buffer;  
    public Produtor(BufferLimitado buffer) {  
        this.buffer = buffer;  
    }  
}
```

# Produtor e Consumidor com Métodos Sincronizados

```
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.getMessage());  
        }  
    }  
}
```

# Produtor e Consumidor com Métodos Sincronizados

```
class Consumidor extends Thread {  
    private BufferLimitado buffer;  
    public Consumidor(BufferLimitado buffer) {  
        this.buffer = buffer;  
    }  
}
```

# Produtor e Consumidor com Métodos Sincronizados

```
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.getMessage());  
        }  
    }  
}
```

# Produtor e Consumidor com Métodos Sincronizados

```
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`)



# 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 Mensagens (Rendezvous)

- 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 Mensagens (Rendezvous)

```
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 Mensagens (Rendezvous)

```
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) &
      " esta na posicao " & Integer'Image(posicao));
```

# Passagem de Mensagens (Rendezvous)

```
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) &
        " esta na posicao " & 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 Mensagens (Rendezvous)

## ■ 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

- 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

# 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>;
```



# Esquema de uma Unidade Protegida Simples

```
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>;
```

# 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;
```

# Implementação de um Objeto Protegido

```
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;
```

# Produtor e Consumidor com Objetos Protegidos

```
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;
```

# Produtor e Consumidor com Objetos Protegidos

```
task type Consumidor is
  entry Iniciar(buf : in ABuffer);
end Consumidor;
end Fabrica;
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;
```

# Produtor e Consumidor com Objetos Protegidos

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

# Produtor e Consumidor com Objetos Protegidos

```
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;
```

# Produtor e Consumidor com Objetos Protegidos

```
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;
```



# Produtor e Consumidor com Objetos Protegidos

```
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
  - ADA oferece troca de mensagens e objetos protegidos