Concorrência e mecanismos de suporte

Disciplina de Modelos de Linguagens de Programação

Concorrência

- Definição: disputa ou cooperação no uso de recurso(s)
- Envolve (tipos de interação):
  - Competição (e.g., MUTEX, transações)
  - Cooperação (e.g., produtor-consumidor)
  - Sincronismo de competição
    (quando um recurso não pode ser usado simultaneamente)
  - Sincronismo de cooperação
    (quando uma necessita aguardar outra para continuar)
- Por que devemos nos preocupar com ela?

Necessidade

- A evolução tecnológica nos permite e a sociedade exige:
  - Simulações e processamento de grandes volumes de dados
  - Concorrência em sistemas monousuário
    - vários softwares executando simultaneamente
      (e.g., e-mail + antivírus + navegação + controle de impressão...)
    - softwares com diferentes tarefas ativas ao mesmo tempo
      (e.g., aplicações web, multimídia, jogos...)
- Aplicações não devem usar a CPU enquanto esperam algum evento ou entrada de dados (busy wait)!

Evolução dos sistemas computacionais

- Sistemas monoprogramados, monousuário
- Sistemas multiprogramados e multiusuário
  - Spooling, interrupted I/O
  - Processos e seus estados, timesharing
  - Redes de computadores
  - Distribuição e paralelismo (real vs simulado)
- Sistemas multicore

Níveis de concorrência

- Nível de instrução de máquina
  - Duas ou mais instruções de máquina sendo executados simultaneamente.
  - Pipelines, processadores vetoriais...
  - Controlado pelo hardware
- Nível de comandos (instrução de programa)
  - Processamento vetorial, operações sobre conjuntos uniformes de dados
  - Máquinas SIMD - Single Instruction Multiple Data
  - Um controlador, pares processador + memória local
- Nível de Unidade
  - Dois ou mais subprogramas sendo executados simultaneamente (em vários processadores ou em um único)
  - Processos ou threads
  - máquinas MIMD (Multiple Instructions Multiple Data)
  - processadores com memória compartilhada ou distribuída
- Nível de Programa
  - Dois ou mais programas executados simultaneamente
  - Processos
  - Controlada pelo sistema operacional (timesharing, escalonamento...)

Execução concorrente

- Execução concorrente não significa execução simultânea!
- A execução de unidades concorrentes admite as seguintes possibilidades:
  - **Pseudoparalela**: execução em um único processador
  - **Paralela**: execução em vários processadores que compartilham memória
  - **Distribuída**: execução em vários processadores independentes, sem compartilhamento de memória
- O programa geralmente não possui controle sobre a ordem e o tempo de execução das unidades concorrentes

Concorrência Lógica e Física

- Concorrência Lógica
  - existe um único processador
  - várias unidades executam de forma intercalada

- Concorrência Física
  - implica paralelismo físico (mais de um processador disponível)
  - execução acontece em mais de um contexto ao mesmo tempo
  - podem existir várias unidades concorrentes do mesmo programa
  - execução paralela e distribuída

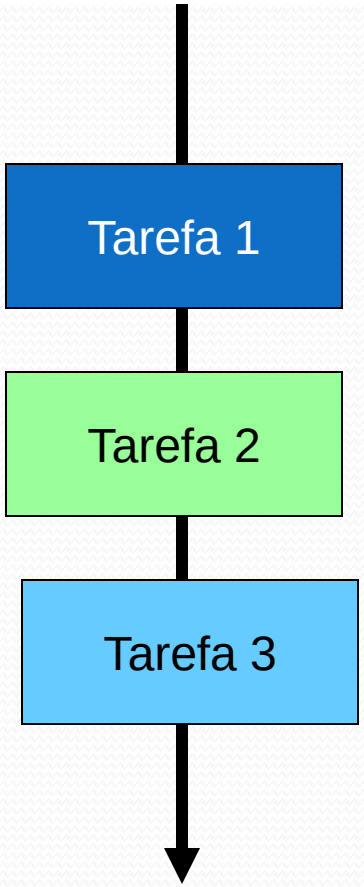
Programação concorrente

- O termo programação concorrente é usado no sentido abrangente, para designar tanto a **concorrência física** quanto a **concorrência lógica**
  - do ponto de vista da LP, a semântica é a mesma
  - diferença no implementador da LP

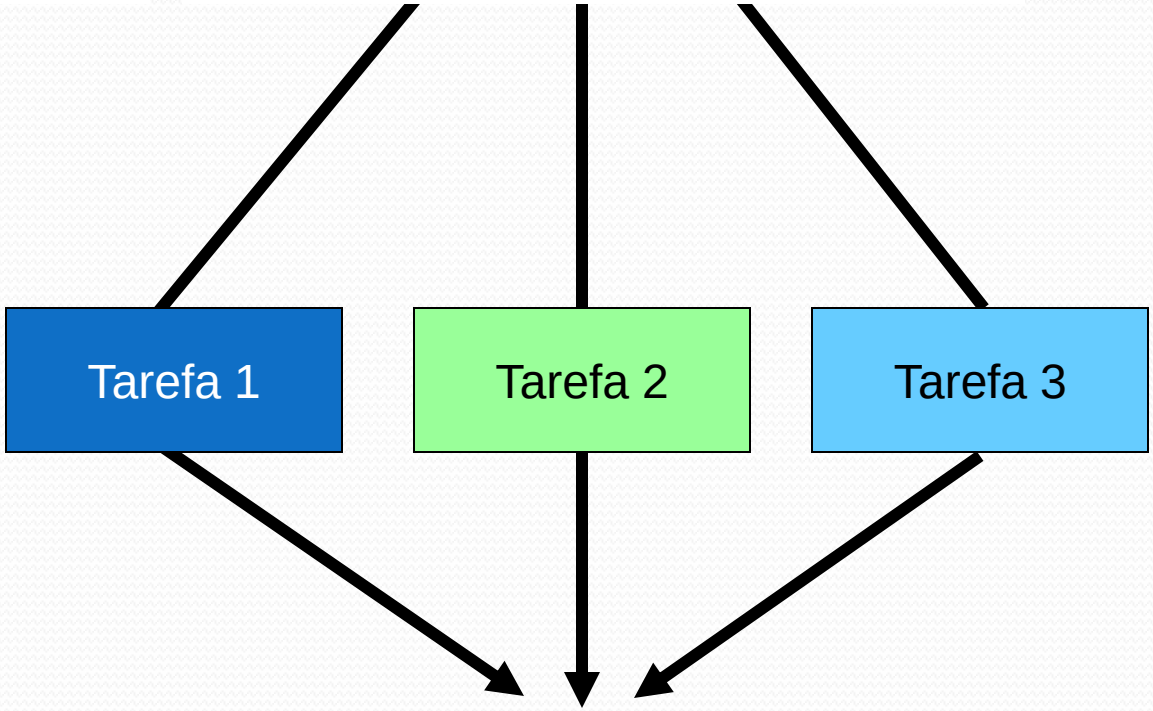
- Uma **unidade concorrente** é um componente de um programa que não exige a execução sequencial
  - sua execução pode ser realizada antes ou após a execução de outros componentes do programa

Fluxo sequencial x concorrente

Fluxo único de execução



Vários fluxos de execução



Cada fluxo possui uma pilha de execução!

Objetivos da programação concorrente

- Reduzir o tempo total de processamento
  - múltiplos processadores
- Aumentar confiabilidade e disponibilidade
  - processadores distribuídos
- Obter especialização de serviços
  - sistemas operacionais
  - simuladores
- Implementar aplicações distribuídas
  - correio eletrônico

Terminologia

- **Tarefa (*task*)**
  - Unidade concorrente
  - Unidade de programa que pode ser executada concorrentemente com outras unidades
  - Cada tarefa pode ser: um conjunto de comandos, um conjunto de sub-rotinas

Tarefas diferem de sub-rotinas:

- podem ser iniciadas implicitamente
- código chamador não precisa esperar o término da tarefa para continuar sua própria execução
- controle pode retornar ou não ao código chamador ao término da tarefa
- tarefas normalmente se comunicam (troca de mensagens , parâmetros ou acesso compartilhado)
- duas ou mais "instâncias ativas" da sub-rotina podem existir (código reentrante)

Terminologia

- ***Thread***
  - linha ou contexto de execução (cada tarefa define uma *thread*)
  - implementadas sobre um ou mais *processos* providos pelo sistema operacional
  - Sistemas Operacionais chamam de *threads* os processos que compartilham o espaço de endereçamento
  - Para algumas LPs, uma *thread* é uma *task*

Mecanismos de programação concorrente

- Compreendem as construções que as linguagens usam para:
  - indicar quais unidades são concorrentes
  - ativar e controlar um fluxo de execução concorrente
  - possibilitar a interação e sincronização entre unidades concorrentes
    - Interação (memória compartilhada ou troca de mensagens)
    - Sincronização (controle da ordem relativa de execução)
    - não necessariamente concordam com o HW implementado
- Podem ser providos por
  - linguagens concorrentes
  - extensões suportadas pelo compilador da linguagem (ver Sun Studio)
  - bibliotecas ou pacotes

Mecanismos de programação concorrente

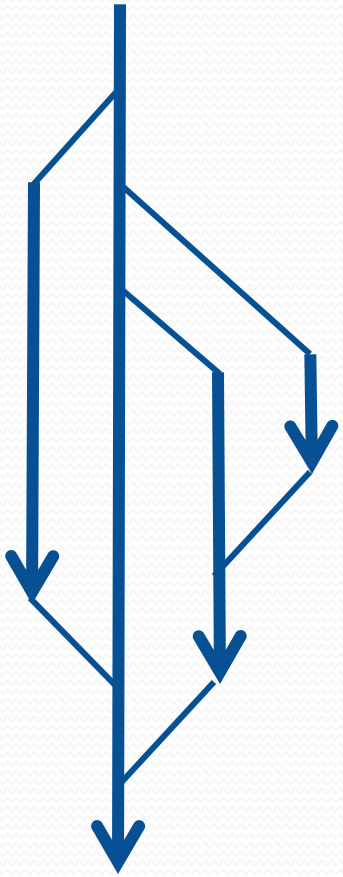
- Através de linguagens concorrentes
  - vantagem do suporte do compilador
  - integração do gerenciamento da criação e comunicação de threads com conceitos de verificação de tipos, escopo e exceções
- Através de bibliotecas ou pacotes
  - e.g., PVM e MPI estão disponíveis para C, C++ e Fortran
  - biblioteca PThread do C/C++ (interface padrão POSIX para *threads*)

Alguns mecanismos

- Comandos Fork e Join (especificação e controle de unidades concorrentes)
- Threads (especificação e controle de unidades concorrentes)
- Semáforos (controle de concorrência em regiões críticas)
- Monitores (controle de concorrência em regiões críticas)
- Mensagens (controle de concorrência)

Mecanismo: Fork e Join

- Mecanismos para ativar e controlar fluxos de execução
- Fork divide o fluxo de execução
- Join integra (espera)
- Funciona em C Unix-like



Exemplo de uso de 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++)
      printf(arg);
   exit(0);
}

int main() {
  int pid;
  int estado_filho;
  pid = fork();
  if(pid<0) { // erro
    perror("erro ao criar 1o filho");
  }else {
    if(pid==0){ // filho
       imprime("-");
    }
  }
  pid = fork();
  if(pid<0){ // erro
    perror("Erro de criação do segundo
filho");
  } else {
    if(pid==0){ // filho
      imprime(".");
    }
  }
  wait(&estado_filho);
  wait(&estado_filho);
  return 0;
}
```

Mecanismo: Threads

- Definições:
  - Fluxo de controle sequencial isolado dentro de um programa
  - *Light weight process*
- Considerações:
  - um único programa pode ter diversas *threads* concorrentes, realizando várias tarefas “ao mesmo” tempo
  - diferentes *threads* podem executar em diferentes processadores, se disponíveis, ou compartilhar um processador único
  - diferentes *threads* no mesmo programa compartilham um ambiente global (memória, processador, registradores, etc.)
- Implementações:
  - PThreads C (POSIX threads)
  - Threads java

Exemplo de uso de PThreads:

```
#include <stdio.h>
#include <string.h>
#include <pthread.h>
void imprime(char *arg) {      // função executada pela thread
   int i;
   for (i=0; i<1000000; i++)
      printf((char*) arg);
   pthread_exit(0);
}
int main() {
  pthread_t tid1, tid2;
  phtread_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;
}
```

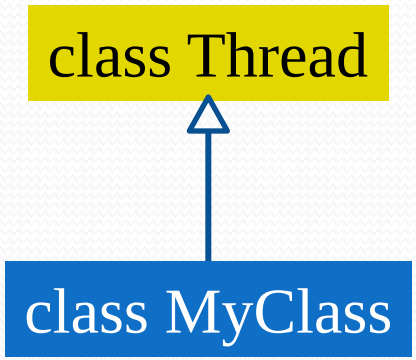
Threads em Java: opções

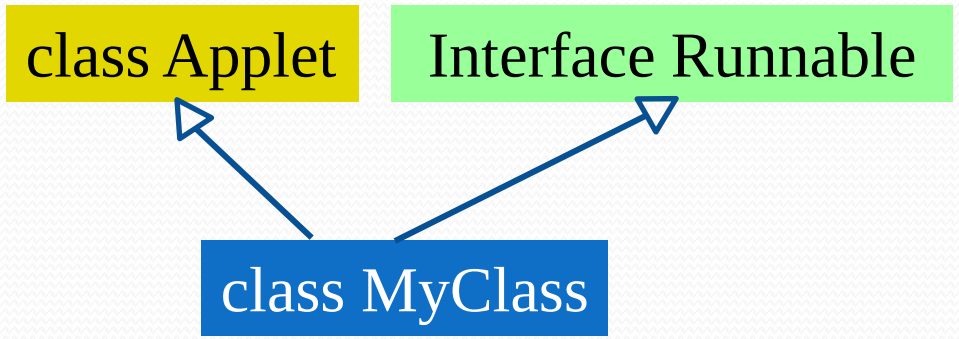
1. Criar uma subclasse da classe Thread

```
public class MyClass
extends Thread { ... }
```

2. Implementar a interface Runnable;

```
public class MyClass
extends Applet implements
Runnable { ... }
```





Threads em Java: execução

- Método **run()** define o que deve ser executado:

```
Class MinhaThread extends Thread{
   public void run( ) {
      for (int count=0; count<1000; count++)
         System.out.println(nome);
   }
}
```

- A atividade concorrente inicia com **start()**:

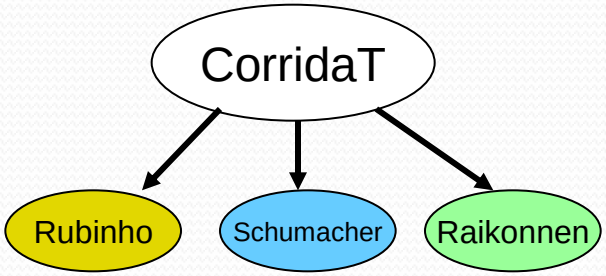
```
public static void main(String[] arg) {
   um.start();
   dois.start();
}
```

Estados de threads

- New
  - thread criado, mas ainda não executando
- Runnable
  - depois de start(), thread está apta a ser executada (depende da disponibilidade do sistema)
- Blocked
  - aguardando algum evento (e.g., I/O ou em decorrência de sleep(), suspend() ou wait())
- Dead
  - execução encerrada (o objeto é destruído)

Exemplo de Thread Java em ação

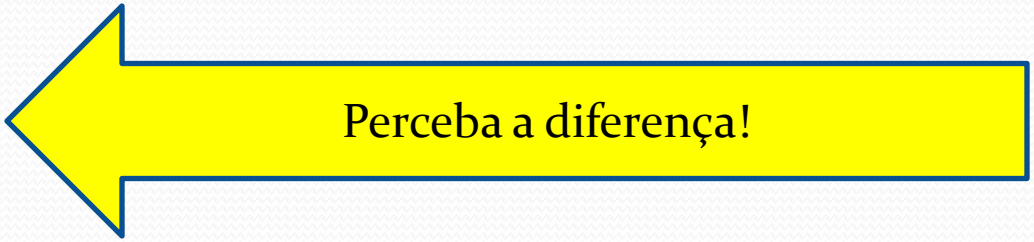
```
class Piloto extends Thread{
   private String nome;

   public Piloto(String str){
      nome = str;
   }

   public void run(){
      system.out.println("****LARGADA ****");
      system.out.println("Primeira volta: " + nome);
      for(int cont=0; cont<10000; cont++){ };
      System.out.println(nome + " -> Terminou a Corrida !!!");
    }
}

public class CorridaT{
   public static void main(String[] args){

       Piloto um = new Piloto("Rubinho");
       Piloto dois = new Piloto("Schumacher");
       Piloto tres = new Piloto("Raikonnen");

       um.start();
       dois.start();
       tres.start();
    }
}
```



Quem terminará antes?

Outro exemplo com Runnable

```
class PilotoR implements Runnable{
   private String nome;
   public PilotoR(String str){
      nome = str;
   }
   public void run(){
      System.out.println("*** LARGADA ***");
      System.out.println(" Primeira volta:" + nome);
      for(int cont=0; cont<10000; cont++) {};
      System.out.println(nome + " -> Terminou a Corrida !!!");
   }
}

public class CorridaR{
   public static void main(String[] args){
      PilotoR um = new PilotoR("Rubinho");
      PilotoR dois = new PilotoR("Schumacher");
      PilotoR tres = new PilotoR(" Raikonnen ");

      new Thread(um).start();
      new Thread(dois).start();
      new Thread(tres).start();
   }
}
```



Threads: alguns métodos relevantes

- **start()**
  - inicia a execução do Thread
- **sleep(ms)**
  - suspende a execução por um tempo determinado (em milissegundos) e automaticamente recomeça a execução
- **wait()**
  - coloca a thread em espera
- **notify()**
  - avisa (acorda) uma thread em espera
- **notifyall()**
  - avisa (acorda) todas as threads em espera

Mais um exemplo de threads

```
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.prinln(">");
      }
      System.out.println("getName()+" completou a prova.");
   }
}
public class Corrida{
   public static void main(String[] args){
         Carro um = new Carro("Rubinho");
         Carro dois = new Carro("Schumacher");
         um.start(); dois.start();
         try { um.join(); } catch(Exception e) { }
         try { dois.join(); } catch(Exception e) { }
   }
}
```

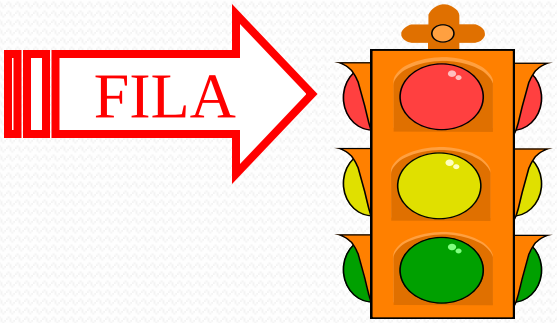
Controle da concorrência: mecanismos

- **Semáforos** (Dijkstra, 1965)
  - Exigem memória compartilhada
  - Podem ser usados para cooperação e competição
  - Adotados em Algol 68 e PL/I (eventos)
- **Monitores** (Brinch-Hansen 1973, Hoarse 1974)
  - Exigem memória compartilhada
  - Baseados em tipos abstrados de dados
  - Adotados em Concurrent PASCAL, MODULA e Java
- **Passagem de mensagens** (Brinch-Hansen e Hoarse 1978)
  - Podem ser usados para programação distribuída
  - Adotado em Ada (*rendez-vous*)

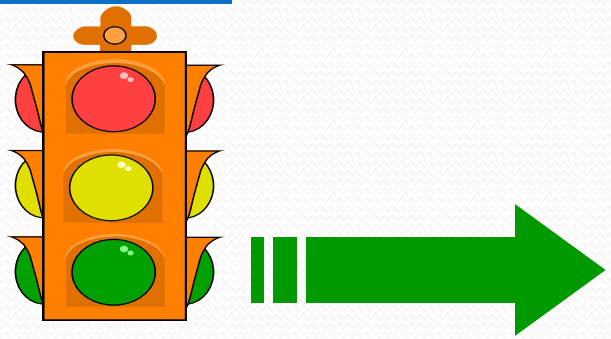
Semáforos

- **Utilizados para implementar guardas no código de acesso a dados compartilhados**
  - devem se assegurar que todas as tentativas de execução do código "guardado" (código ou região crítica) sejam realmente executadas em algum momento
  - possuem mecanismo para anotar e recuperar tentativas que não executaram ainda

Código (ou região) crítico



Código de acesso a recursos compartilhados

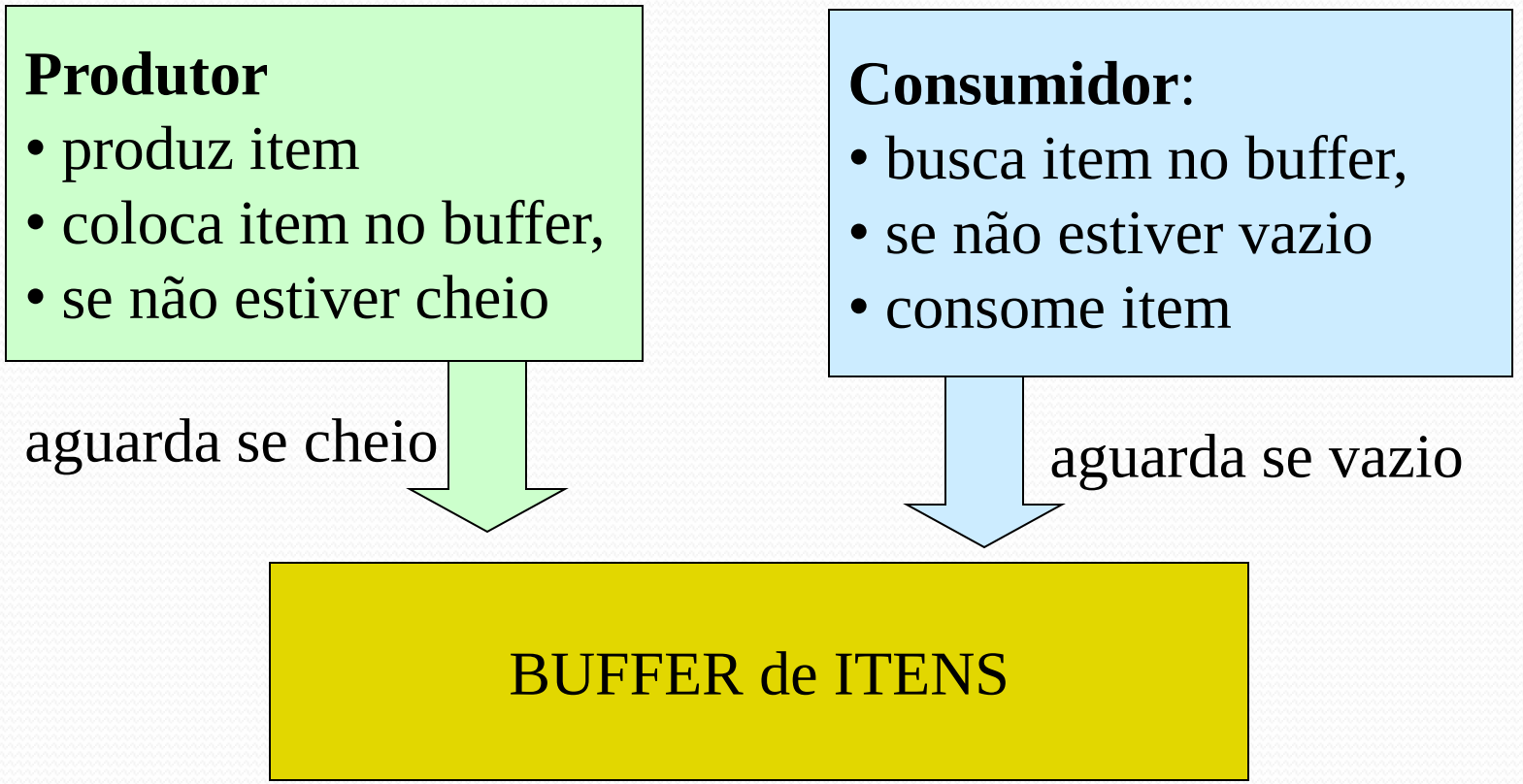


Semáforos

- **Um semáforo é uma estrutura de dados que consiste em um contador e numa fila de tarefas**
- **Possuem apenas duas operações**:
  - Aguarda / Bloqueia: primitiva P
    (P = *passeren* , ou seja, passar)
  - Continua / Libera: primitiva V
    (V = *Vrygeren* , ou seja, liberar)
- Primeira função: comunicação por competição
- Generalização para cooperação

Semáforos: exemplo de aplicação

- **Problema do produtor versus consumidor**



Atenção sobre a seção crítica

- Buffer é a estrutura compartilhada
- Operações no buffer:
  - Escrita
  - Leitura
- Problemas:
  - Ler e escrever em uma posição simultaneamente (consistência do dado)
  - Ler de um buffer vazio (*underflow*)
  - Escrever em um buffer cheio (*overflow*)

Semáforos: exemplo de aplicação

```
Process Produtor;
Var i: integer;
Begin
   loop
      produz(i);

      coloca(i);

   end loop;
End;
```


```
Process Consumidor;
Var i: integer;
Begin
   loop
      retira(i);

      consome(i);

   end loop;
End;
```

Problema: 2 unidades concorrentes acessando a região crítica

Semáforos: exemplo de aplicação

```
Process Produtor;
Var i: integer;
Begin
   loop
      produz(i);
      P(exclusão);
      coloca(i);
      V(exclusão);
   end loop;
End;
```


```
Process Consumidor;
Var i: integer;
Begin
   loop
      retira(i);
      P(exclusão);
      consome(i);
      V(exclusão);
   end loop;
End;
```

Solução: sincronismo de competição utilizando semáforo binário

Semáforos: exemplo de aplicação

```
Process Produtor;
Var i: integer;
Begin
   loop
      produz(i);
      P(exclusão);
      coloca(i);
      V(exclusão);
   end loop;
End;
```


```
Process Consumidor;
Var i: integer;
Begin
   loop
      retira(i);
      P(exclusão);
      consome(i);
      V(exclusão);
   end loop;
End;
```

Problema: evitar consumo de recursos de forma desnecessária

Solução por busy-wait

- Algoritmo genérico:

```
Condicao = falso
While (condicao_não_for_verdadeira){ ... }
```

- Exemplo:

```
bufferVazio = true
While (bufferVazio) { sleep(10); }
```

- Problema:
  - ocupa o processador sem computar nada
  - troca de contexto versus desempenho
  - não estruturado

Semáforos: exemplo de aplicação

```
Process Produtor;
Var i: integer;
Begin
   loop
      produz(i);
      P(naocheio);
      coloca(i);
      V(naovazio);
   end loop;
End;
```


```
Process Consumidor;
Var i: integer;
Begin
   loop
      retira(i);
      P(naovazio);
      consome(i);
      V(naocheio);
   end loop;
End;
```

Solução: sincronismo de cooperação com 2 semáforos

Semáforos: exemplo de aplicação

```
Process Produtor;
Var i: integer;
Begin
   loop
      produz(i);
      P(naocheio);
      P(exclusão);
      coloca(i);
      V(exclusão);
      V(naovazio);
   end loop;
End;
```


```
Process Consumidor;
Var i: integer;
Begin
   loop
      retira(i);
      P(naovazio);
      P(exclusão);
      consome(i);
      V(exclusão);
      V(naocheio);
   end loop;
End;
```

Solução: sincronismo de cooperação e competição
(3 semáforos)

Semáforos: problema de deadlock

```
Process Produtor;
Var i: integer;
Begin
   loop
      produz(i);
      P(naocheio);
      P(exclusão);
      coloca(i);
      V(exclusão);
      V(naovazio);
   end loop;
End;
```

```
Process Consumidor;
Var i: integer;
Begin
   loop
      retira(i);
      P(exclusão);
      P(naovazio);
      consome(i);
      V(exclusão);
      V(naocheio);
   end loop;
End;
```

Inverter ordem

Esquecer um dos comandos

Semáforos: avaliação

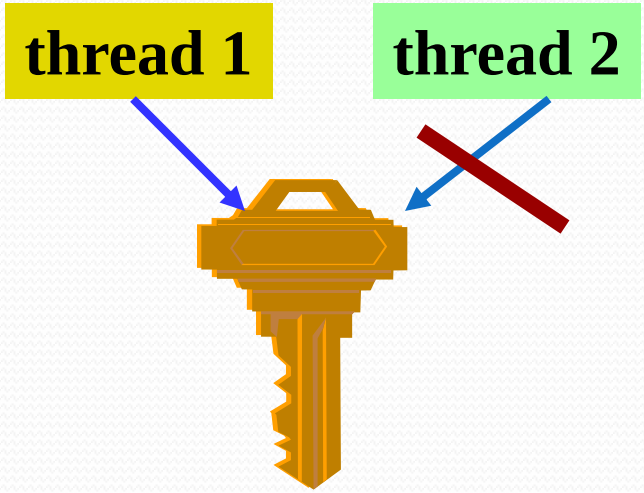
- Operações nos semáforos devem ser atômicas
  - Não podem ser interrompidas
  - Processador normalmente tem instruções atômicas
- Factível, mas inseguro
- Compilador não tem como verificar se a localização das chamadas de P e V estão corretas

Monitores

- São entidades que controlam o acesso a um recurso encapsulado, através de métodos específicos
- Nenhum método que esteja dentro do monitor pode ser chamado por mais de um processo ou thread simultaneamente
- Fica a cargo do compilador a implementação da exclusão mútua
- Exemplo: uma classe em O.O.

Método syncronized em java

- Cada objeto que possui um método synchronized é considerado um monitor
- Quando este método é chamado ocorre um bloqueio (*lock*)
- Todos os demais threads que desejam executar algum método deste objeto devem aguardar o desbloqueio



public synchronized void recurso (..){.....}

OBS: Uma classe pode possuir diversos métodos synchronized mas somente um pode estar ativo em um determinado momento

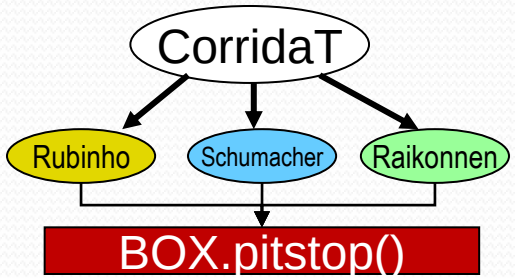
Lista de espera

- Objetos monitores mantém uma lista de threads que aguardam a execução de algum método synchronized
- Nesta lista são inseridas threads que :
  - chamaram um método synchronized
  - chamaram um wait (aguardam notify)
- Tratamento de Deadlock: se algum método synchronized não for desbloqueado, o mecanismo de sincronização Java dispara uma exceção e o tratador desbloqueia o método

Exemplo: corrida com sincronismo

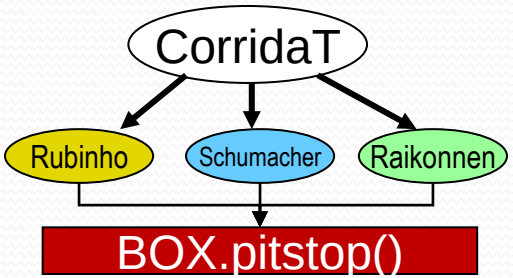
```
class Box {
   private int tempo=50000;
   Box(){} // construtor
   public synchronized void pitStop(String nome){
      System.out.println(nome + " no box");
      for (int i=1;i<tempo;i++){} // durante pitstop
         notify();
   }
}
class Exemplo {
   public static void main(String args[]){
      Box b = new Box();   // região crítica
      PilotoBox um   = new PilotoBox("Rubinho",b);
      PilotoBox dois = new PilotoBox("Schumacher",b);
      PilotoBox tres = new PilotoBox("Montoya",b);
      um.start();
      dois.start();
      tres.start();
   }
}
```

Sincronização por competição!



Exemplo: corrida com sincronismo

```
class PilotoBox extends Thread{
  private String nome;
  Box b;
  public PilotoBox(String str, Box b){
     nome   = str;
     this.b = b;
  }
  public void run(){
    system.out.println("*** LARGADA " + nome + "***");
    for (int i=1;i<50000;i++){} // antes do pitstop
    System.out.println(nome + " solicitou pitStop”);

    b.pitStop(nome); // transparente para quem usa!

    for (int i=1;i<50000;i++){} // após pitstop
    System.out.println(nome + " -> Terminou a Corrida !!!");
  }
}
```



Monitores: avaliação

- Sincronismo para competição é implícito (dentro do monitor)
- Sincronismo de comunicação:
  - Estrutura **queue** (fila) pré-definida com a mesma lógica dos semáforos
  - Também herda os mesmos problemas (mas algumas linguagens minimizam alguns deles)
- O que fazer quando processadores usam memória não compartilhada? Monitores seriam solução fácil?

Sincronização de cooperação

- Em Java, é realizada pelos métodos wait e notify (definidos em Object):

```
try{
  while(!condicao) wait();
  /* comandos necessários de serem feitos depois que
  a condição acontecer */
}catch(InterruptedException e) { ... }
```

  - OBS: interruptedexception é gerada em caso de deadlock.

- Ver exemplo do produtor-consumidor com sincronização de cooperação

Mensagens (encontros)

- Modelo geral para comunicação concorrente
  - modela semáforos e monitores
  - usada para cooperação e competição
- Idéia: uma tarefa T1 manda uma mensagem para T2 solicitando um encontro que somente ocorrerá quando T2 estiver apta. O encontro é denominado "rendez-vous"
- Sistema operacional ou biblioteca oferece mecanismos para troca de mensagens síncronas (bloqueantes) ou assíncronas
  - Send
  - Receive
- Mensagens podem ser direcionadas a um processo específico a uma caixa postal (compartilhada)
- Outras formas: variáveis não locais, passagem de parâmetros (chamada explícita com RPC, RMI...)
- Ver: CC++, PVM, MPI, Sun Studio C++

Concorrência: alguns problemas

- Dead-lock:
  - Quando dois processos ficam trancados, um esperando um recurso do outro
- Starvation
  - Quando um processo não obtém acesso a um recurso (não é executado, por exemplo), por sua prioridade ser baixa demais