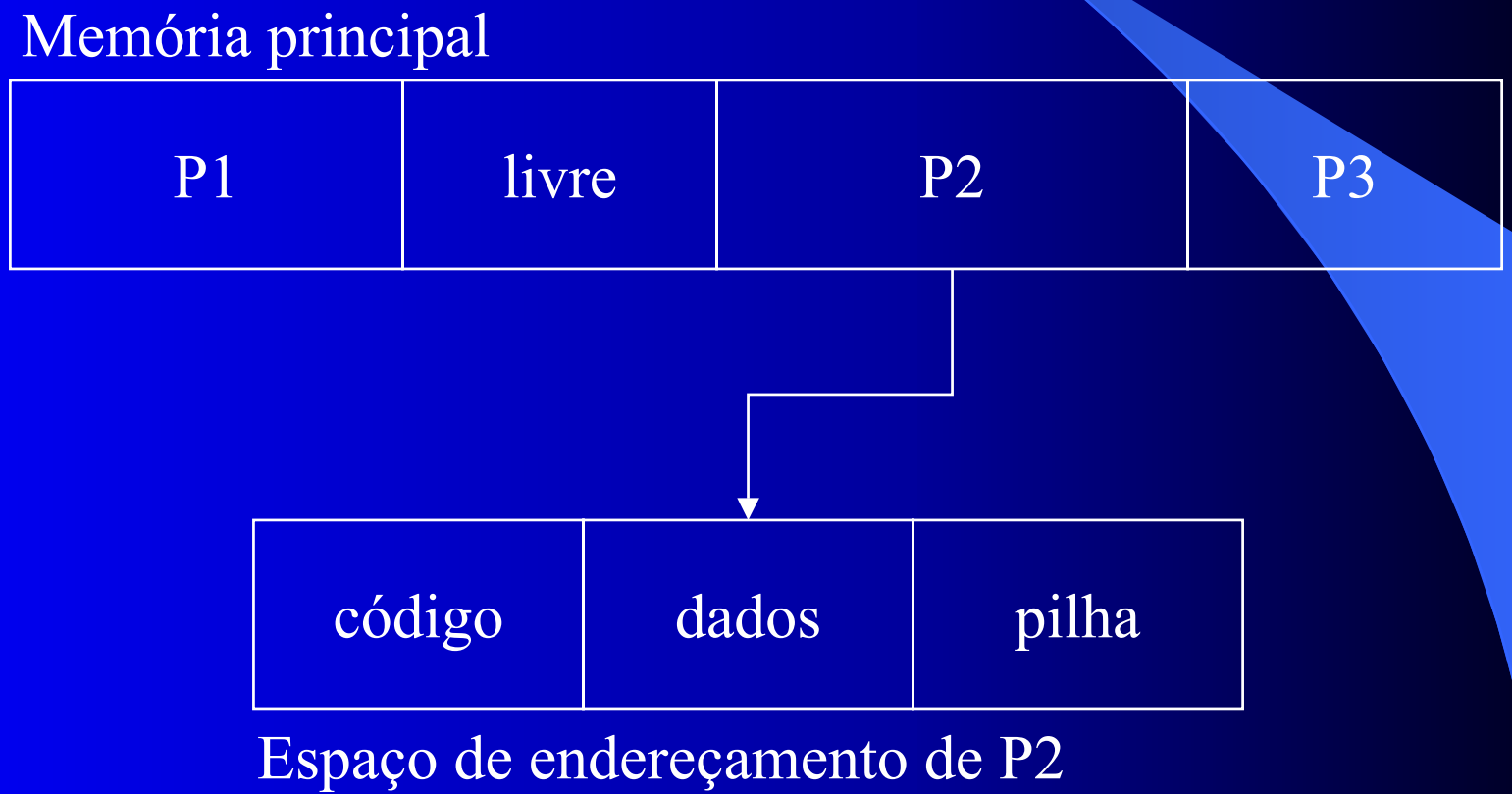
Java Middle-Tier: Componentes WEB

Programação Concorrente

Tecgraf/PUC-Rio

Processos & Memória



Threads ou LWP

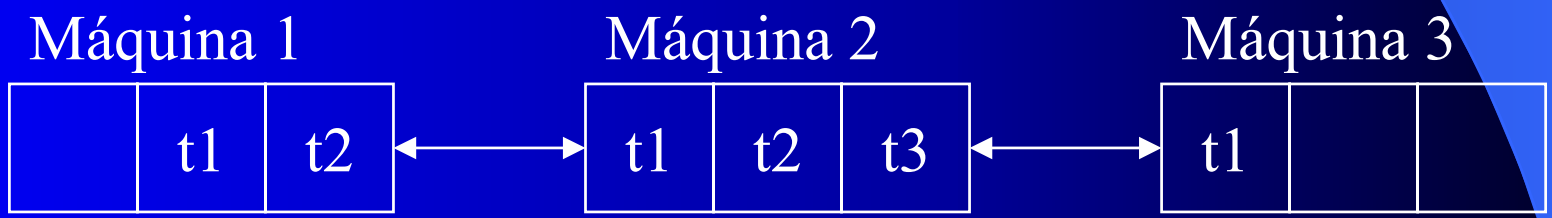
- Processos que compartilham espaço de endereçamento:

P1	P2	P3
threads P1a, P1b, P1c	thread P2a	threads P3a, P3b

- Uma aplicação concorrente é composta por vários *threads* (várias sequências de execução)

Concorrência & Distribuição

- Facilidades de concorrência são importantes para a criação de uma aplicação distribuída

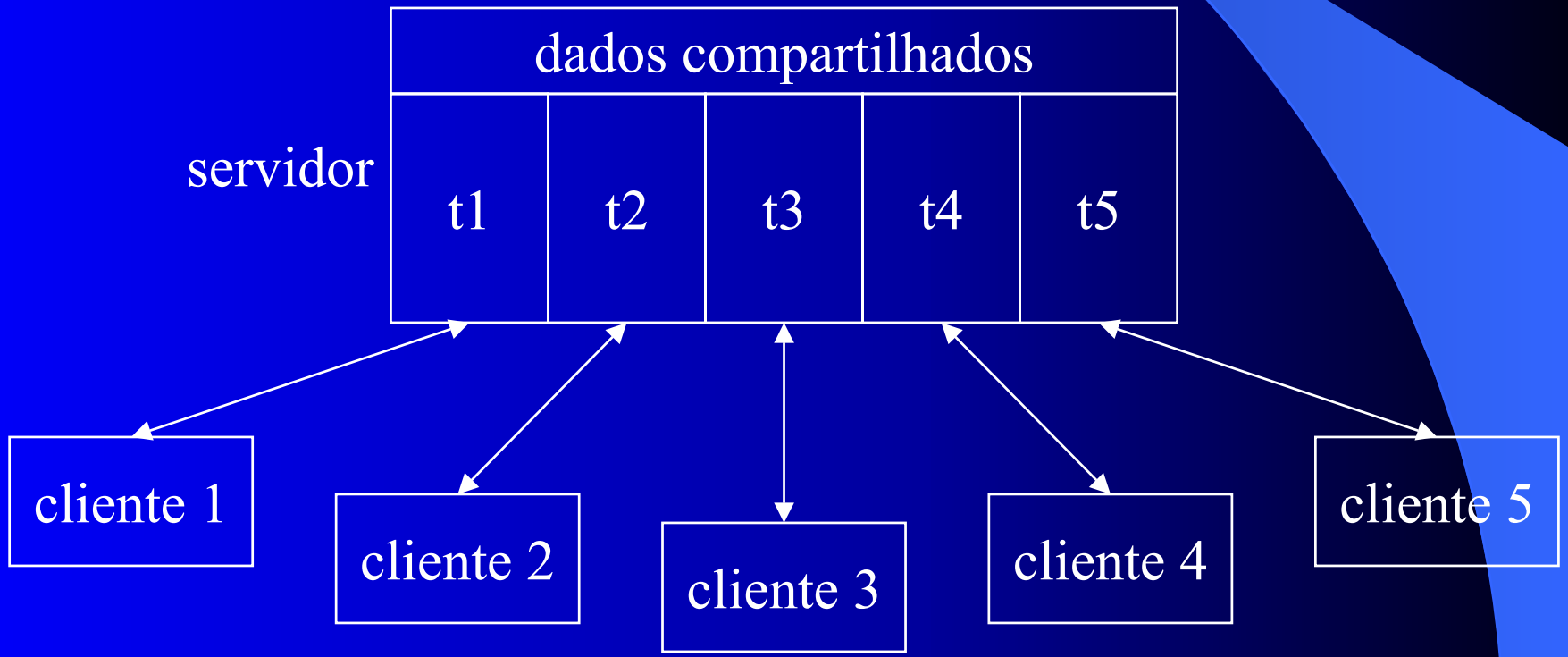


Concorrência & Distribuição

- Muitos dos problemas de programação concorrente reaparecem na programação distribuída

Concorrência em arquitetura cliente-servidor: servidor

- Atendimento simultâneo a vários clientes



Concorrência em arquitetura cliente-servidor: cliente

- Melhor estrutura da aplicação:
  - resposta a eventos de interface e de rede
- Melhor aproveitamento do tempo:
  - disparo de diversas solicitações simultâneas
  - tratamento local de dados enquanto espera resultado de solicitação

Threads em Java

- Uma classe que estende a classe **Thread** pode ser usada para definição de *threads* de controle
- Ao criar um novo objeto dessa classe, cria-se um *thread*
- O início de execução ocorre com a chamada ao método **start**
- O método **run** deve ser redefinido na nova classe para especificar a execução do novo *thread*

Exemplo: ThreadsDorminhocas

```
public class ThreadsDorminhocas {
  public static void main(String[] args) {
    new ThreadDorminhoca("1");
    new ThreadDorminhoca("2");
    new ThreadDorminhoca("3");
    new ThreadDorminhoca("4");
  }
}
```

ThreadDorminhoca

```
class ThreadDorminhoca extends Thread {
  int tempo_de_sono;
  public ThreadDorminhoca(String id) {
    super(id);
    tempo_de_sono = (int) (Math.random() * 5000);
    System.out.println("Tempo de sono da thread "+id+
                       ": "+tempo_de_sono+"ms");
    start();
  }
  public void run() {
    try {
      sleep(tempo_de_sono);
    } catch(InterruptedException exception) {
      System.err.println(exception);
    }
    System.out.println("thread "+getName()+" acordou!");
  }
}
```

Comunicação por memória compartilhada

- Threads concorrentes acessam informações comuns
- Necessidade de garantir a consistência do estado das informações acessadas
  - exclusão mútua
- Necessidade de espera bloqueada
  - cooperação

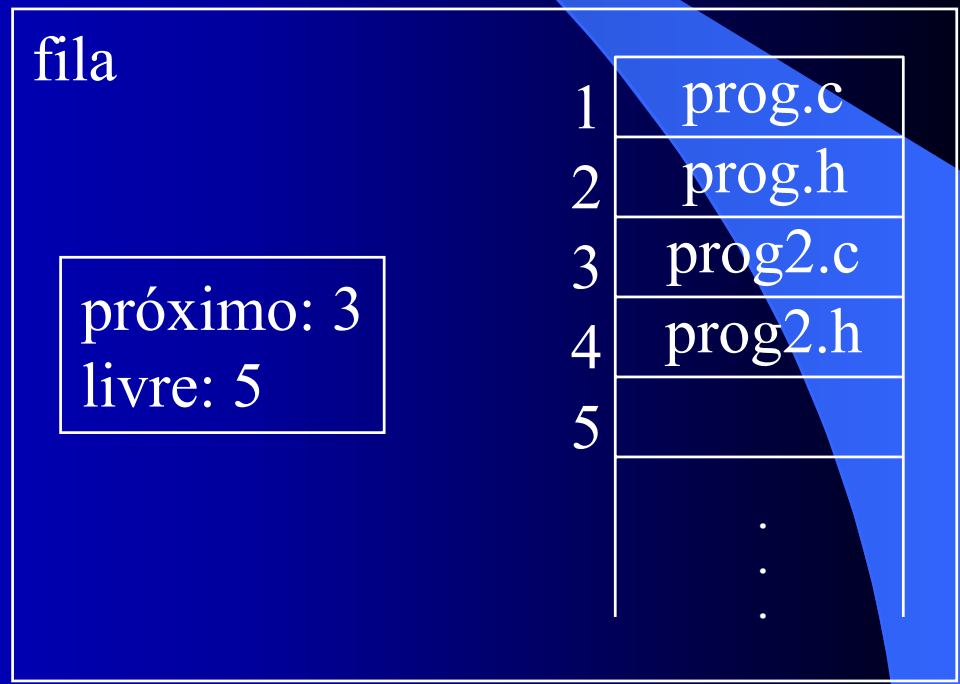
↳ **Sincronização!**

Exemplo: consistência

- Dois *threads* colocando itens em uma fila

thread A:
```
fila.poenafila(item1);
```

thread B:
```
fila.poenafila(item2);
```



Exemplo: acesso a saldo bancário

```
class Conta {
  private int saldo;
  public Conta (int ini) {
    saldo = ini;
  }
  public int veSaldo() {
    return saldo;
  }
  public void deposita(int dep) {
    for (int i=0; i<dep; i++) {
      try {
        Thread.sleep(10); // só pra dar chance ao escalonador!
      }
      catch(InterruptedException exception) {
        System.err.println(exception);
      }
      saldo++;
    }
  }
}
```

Acesso a saldo bancário (cont)

```
public class ThreadsEnxeridas {
  public static void main(String[] args) {
    int repet = 20;
    Conta cc = new Conta(0);
    (new ThreadEnxerida("1", cc, repet)).start();
    (new ThreadEnxerida("2", cc, repet)).start();
    (new ThreadEnxerida("3", cc, repet)).start();
    (new ThreadEnxerida("4", cc, repet)).start();
  }
}
```

Acesso a saldo bancário (cont)

```
class ThreadEnxerida extends Thread {
  private int vezes;
  private Conta cc;
  public ThreadEnxerida(String id, Conta a, int qtas) {
    super(id);
    cc = a;
    vezes = qtas;
  }
  public void run() {
    for (int i=0; i<vezes; i++) {
      System.out.println("thread" + getName() +
                         "-  saldo: " + cc.veSaldo());
      cc.deposita(100);
    }
  }
}
```

Captura de estados inconsistentes

```
public void deposita(int dep) {
  for (int i=0; i<dep; i++) {
    saldo++;
  }
}
```

```
public int veSaldo() {
  return saldo;
}
```

Condições de corrida

- Situação onde o resultado final depende do momento em que ocorrerem as trocas de contexto

```
// gera arquivo
Arquivo arq;
...
```

trecho sem condição de corrida

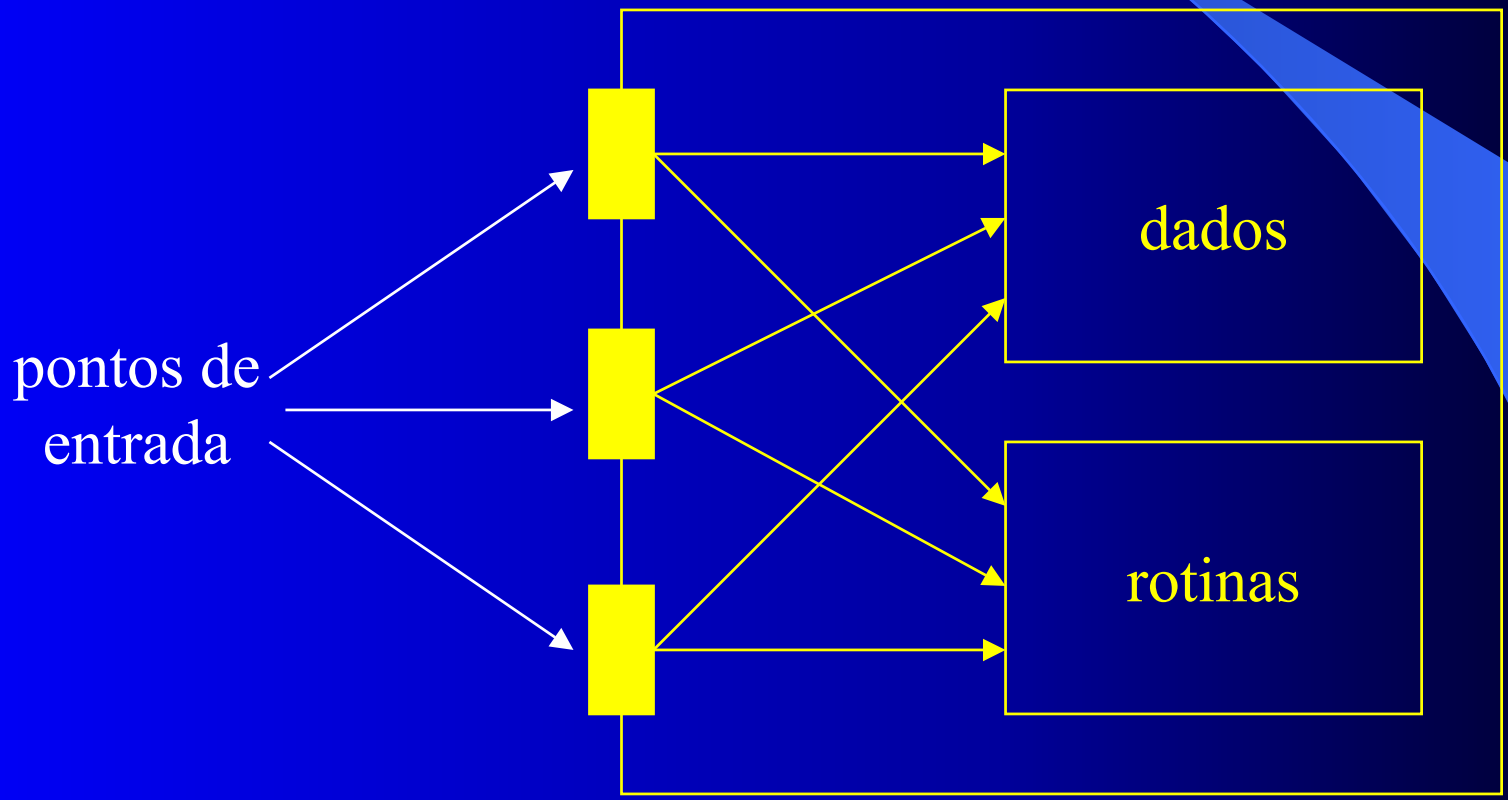
```
// Coloca na fila
int p = fila.livre;
fila.insere(arq, p);
fila.livre++;
```

trecho com condição de corrida (**Região Crítica**)

Exclusão mútua

- Maneira de se evitar que dois processos entrem nas suas regiões críticas correspondentes ao mesmo tempo
- Em Java, uso de monitores
  - métodos declarados como **synchronized** garantem que nenhuma outra chamada ao método pode executar concorrentemente

Monitor

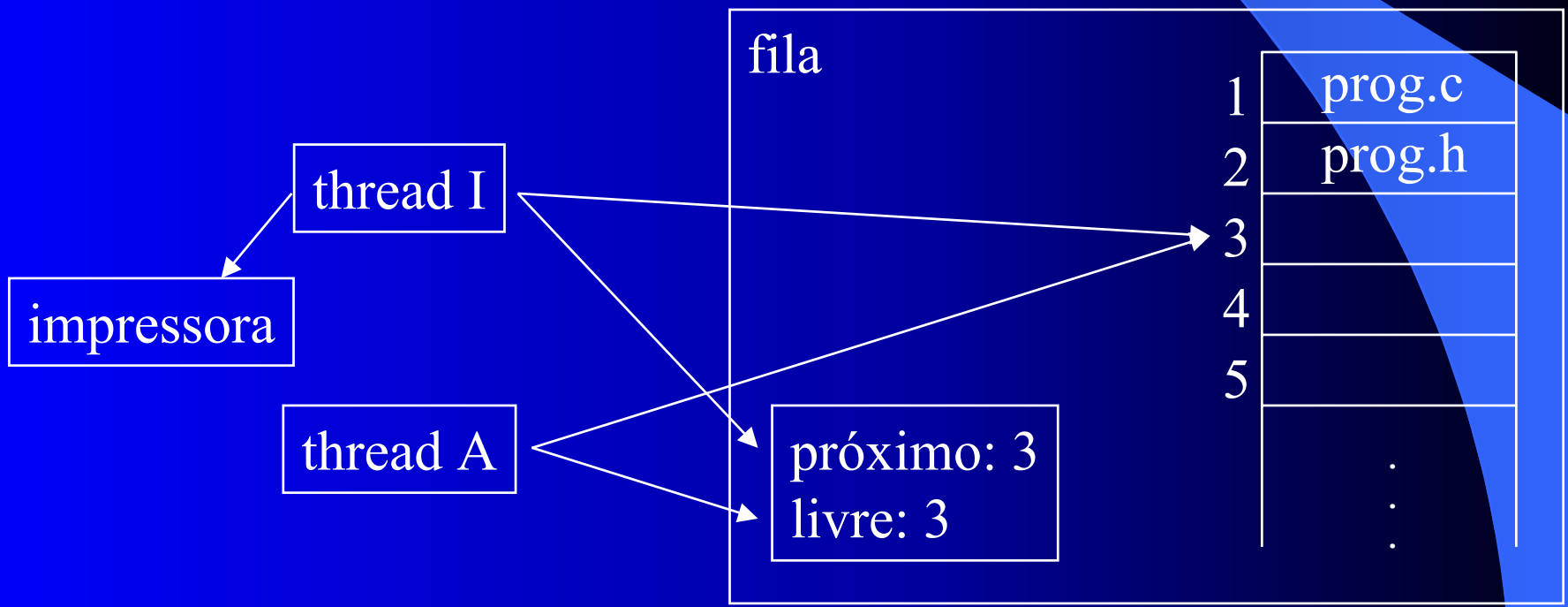


Exemplo de uso: acesso a saldo bancário

```
class Conta {
  private int saldo;
  public Conta (int ini) {
    saldo = ini;
  }
  public synchronized int veSaldo() {
    return saldo;
  }
  public synchronized void deposita(int dep) {
    for (int i=0; i<dep; i++) {
      try {
        Thread.sleep(10); // só pra dar chance ao escalonador!
      }
      catch(InterruptedException exception) {
        System.err.println(exception);
      }
      saldo++;
    }
  }
}
```

Espera bloqueada

- O *thread* de impressão só pode agir se houver algo para ser impresso:



Soluções para espera bloqueada

- Espera ocupada: o *thread* fica testando continuamente se a condição desejada é satisfeita

```
while (fila.ninguemNaFila());
item = fila.pegaProximo();
```

- É interessante que um *thread* que está a espera de um recurso não desperdice tempo de CPU
  - além disso, um outro *thread* pode mudar o valor da condição entre uma chamada e outra
  - se o método que engloba este código é sincronizado, a condição pode nunca ser alterada

Mais soluções para espera bloqueada

- Criação de mecanismos de bloqueio
  - *thread* "dorme" até que o recurso que ele requer se torne disponível

```
item = fila.pegaProximo();
```

Espera por condição

- Operação de espera permite a um *thread* colocar-se em estado bloqueado até que uma condição seja satisfeita
  - **wait**
- Operação de sinalização permite a um *thread* avisar aos demais a ocorrência de uma condição
  - **signal** ou **notify**

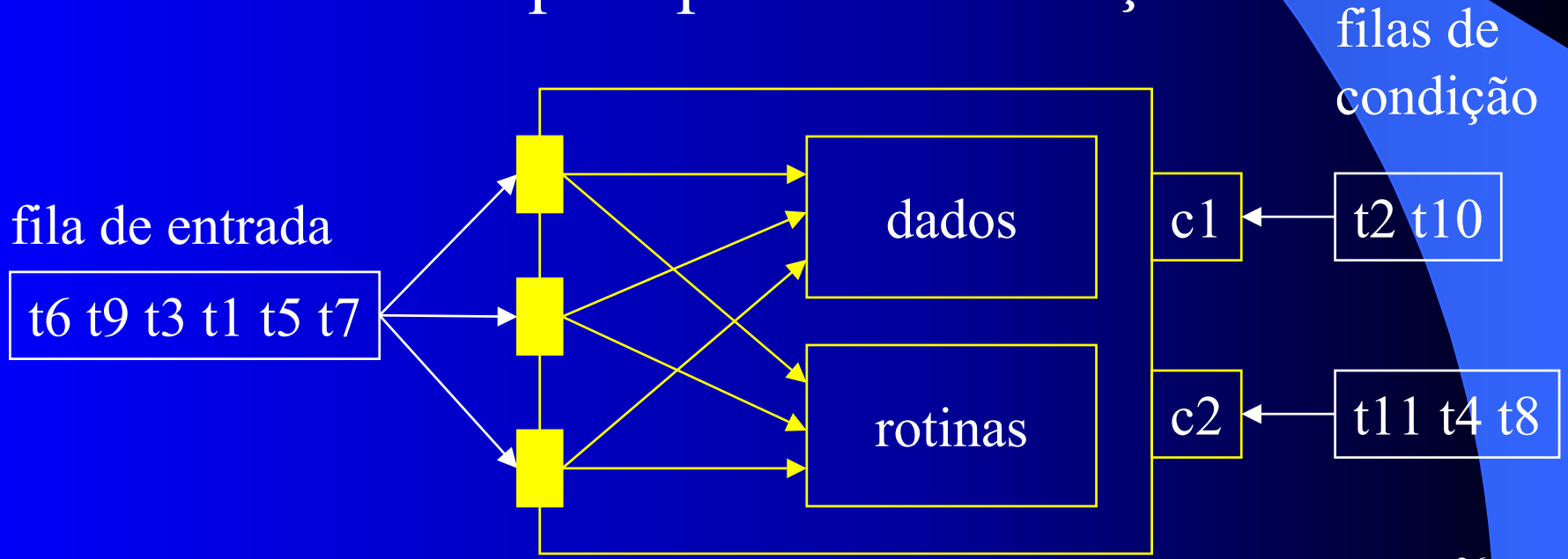
Exemplo: espera bloqueada

```
class fila extends …
  condition ALGUEM;  // variável de condição
  ...
  private int[] buffer = new int[tam];
  ...
  public synchronized int pegaProximo(){
    int i;
    if (proximo==livre) wait(ALGUEM);
    i = buffer[proximo];
    proximo = incmodulo(proximo);
    return i;
  }
  public synchronized void poenafila (int i) {
    buffer[livre] = i;
    livre = incmodulo(livre);
    signal(ALGUEM);
  }
}
```

Não é Java!

Exclusão mútua e condições

- Tipicamente um monitor tem dois tipos de fila: a fila de espera para entrar no monitor e as filas de espera por uma condição



Escalonamento

- Um *thread* que se bloqueia por uma condição libera o acesso de outro ao monitor
- Ao ser sinalizado, um *thread* que estava na fila de condição volta a disputar a exclusão mútua
  - resolvido de diferentes formas em diferentes linguagens

Em Java...

- A classe **Object** define os métodos **wait**, **notify** e **notifyAll**
  - têm que ser chamados de dentro de código sincronizado
- Não existem variáveis de condição
  - **wait** e **notify** agem sobre fila única
    - ao "acordar", um *thread* não pode saber se a condição que ele estava esperando realmente vale
    - uso de **notifyAll**: todos os *threads* bloqueados são desbloqueados e todos devem testar a condição desejada

Object: métodos **wait** e **notify**

- `public final void wait(long timeout) throws InterruptedException`
  - bloqueia *thread* até **timeout** decorrer ou notificação
  - **timeout** em milisegundos
  - se **timeout=0**, bloqueia até notificação
- `public final void wait() throws InterruptedException`
  - equivalente a **wait(0)**
- `public final void notify()`
  - notifica apenas um *thread* bloqueado por condição
- `public final void notifyAll()`
  - notifica todos os *threads* bloqueados por condição

Revendo o exemplo

```
class fila extends ... {
  ...
  private int[] buffer = new int[tam];
  ...
  public synchronized item pegaProximo(){
    item i;
    while (proximo==livre) wait();
    // método sincronizado garante que a
    // condição permanece verdadeira até a
    // execução do próximo comando!
    i = buffer[proximo];
    proximo = incmodulo(proximo);
    return i;
  }
  public synchronized void poenafila (item i) {
    buffer[livre] = i;
    livre = incmodulo(livre);
    notifyAll();
  }
}
```

Ainda não é Java!

Exceções

- O método **interrupt** pode ser chamado por outro *thread*, causando um retorno de **wait** com sinalização de exceção
  - chamada de **wait** tem que tratar essa exceção

Revendo o exemplo

```
class fila extends ... {
  static final int tam = 20;
  private int[] buffer = new int[tam];
  int proximo, livre;
  ...
  public fila() {
    proximo = 0;
    livre = 0;
  }
  public synchronized int pegaProximo(){
    int i;
    while (proximo==livre)
      try {wait();}
      catch (InterruptedException ex) {return -1;}
    i = buffer[proximo];
    proximo = incmodulo(proximo);
    return i;
  }
  public synchronized void poenafila (int i) {
    buffer[livre] = i;
    livre = incmodulo(livre);
    notifyAll();
  }
}
```

É Java!

Exemplo Clássico

- O exemplo do "*buffer* limitado" é um clássico no estudo de concorrência
- A fila que acabamos de ver, implementada por um *array*, se encaixa neste exemplo
- Falta programar o que acontece quando o *array* está cheio

Buffer Limitado

```
class fila extends ... {
  private static final int tam = 20;
  private int[] buffer = new int[tam];
  int proximo, livre;
  ...
  public fila() {
    proximo = 0;
    livre = 0;
  }
  public synchronized int pegaProximo(){
    int i;
    while (proximo==livre)
      try {wait();} // espera ter algum item!
      catch (InterruptedException ex) {return -1;}
    i = buffer[proximo];
    proximo = incmodulo(proximo);
    notifyAll();
    return i;
  }
  public synchronized void poenafila (int i) {
    while (incmodulo(livre)==proximo)
      try {wait();} // espera espaço para item!
      catch (InterruptedException ex) {return;}
    buffer[livre] = i;
    livre = incmodulo(livre);
    notifyAll();
  }
}
```

Métodos de **Thread**

- `public native synchronized void start()`
  - inicia o *thread*
- `public final native boolean isAlive()`
  - retorna **true** se o *thread* iniciou e ainda não terminou
- `public static native void sleep(long millisec) throws InterruptedException`
  - suspende a execução durante o tempo especificado
- `public final void setPriority(int newPriority)`
  - valores entre **Thread.MIN_PRIORITY** e **Thread.MAX_PRIORITY**

Sincronização

- O bloqueio de espera por exclusão mútua ocorre entre métodos sincronizados
  - um método não sincronizado pode ser ativado durante a execução de um método sincronizado
- Muitas operações primitivas de Java, como atribuições e acessos a valores escalares, são *atômicas*
  - há exceções (perigoso contar com essa propriedade)

Comandos sincronizados

- A sincronização pode ser feita em granulidade menor que um método inteiro, através de comandos sincronizados

```
synchronized (expr) comando
```

- A expressão entre parênteses deve retornar um objeto, sobre o qual é colocado um bloqueio
  - esse bloqueio não impede acessos a este objeto, a menos que sejam protegidos por outro comando sincronizado pelo mesmo objeto

Exemplo de comando sincronizado

- Bloqueio sobre um *array*

```
public static void abs (int[] values) {
  synchronized (values) {
    for (int i=0; i<values.length; i++) {
      if (values[i] < 0)
        values[i] = -values[i];
    }
  }
}
```

  - normalmente, o uso de métodos sincronizados é mais simples e mais seguro

Equivalência entre métodos e comandos sincronizados

```
public synchronized void f() {
   ...
}
```

≡

```
public void f() {
   synchronized (this) {
   }
}
```

Mais sobre sincronização

- A declaração de sincronização só pode ser feita em classes, não em interfaces
  - propriedade de implementação
- Um método sincronizado pode chamar outro método sincronizado
- Construtores não podem ser sincronizados

Escalonamento

- Não há garantias sobre os momentos em que ocorre escalonamento
- *Threads* de prioridade mais alta sempre têm preferência sobre os demais
  - *threads* de prioridade menor sofrem preempção em favor de *threads* com maior prioridade
- Não há garantia de preempção para escalonamento entre *threads* de mesma prioridade
  - uso de **sleep** no exemplo

Soluções com sincronização

- Propriedades desejáveis:
  - garantia contra inconsistências
  - possibilidade de concorrência máxima
  - ausência de *deadlock* e *starvation*

Exclusão mútua e espera bloqueada: problemas

- *Deadlock*: um conjunto de processos fica bloqueado, cada um a espera de um recurso que o outro detém
- *Starvation*: alguns processos são repetidamente preteridos, enquanto outros ficam com o acesso ao recurso desejado
- Exemplo: Considere uma estrada com uma ponte por onde só passa um carro de cada vez

Deadlock

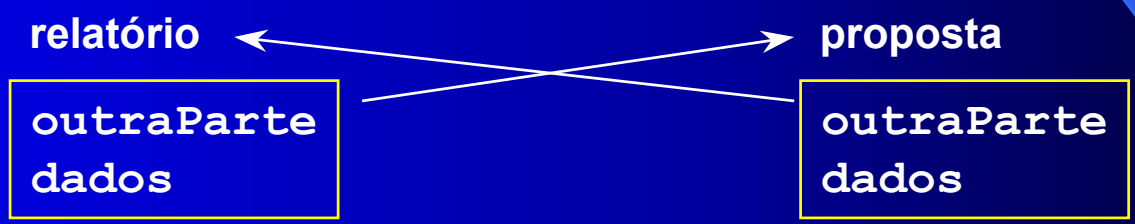
- Um carro começa a atravessar de cada um dos lados e os dois se encontram no meio da ponte
- Nenhum dos dois tem como continuar

Exemplo de *deadlock*

```
class Documento {
  DadosDoc dados; // dados do documento
  Documento outraParte;
  ...
  synchronized void imprime() {
    // imprime infos em dados
  }
  synchronized void imprimeTudo() {
    // imprime todas as partes
    outraParte.imprime();
    imprime();
  }
}
```

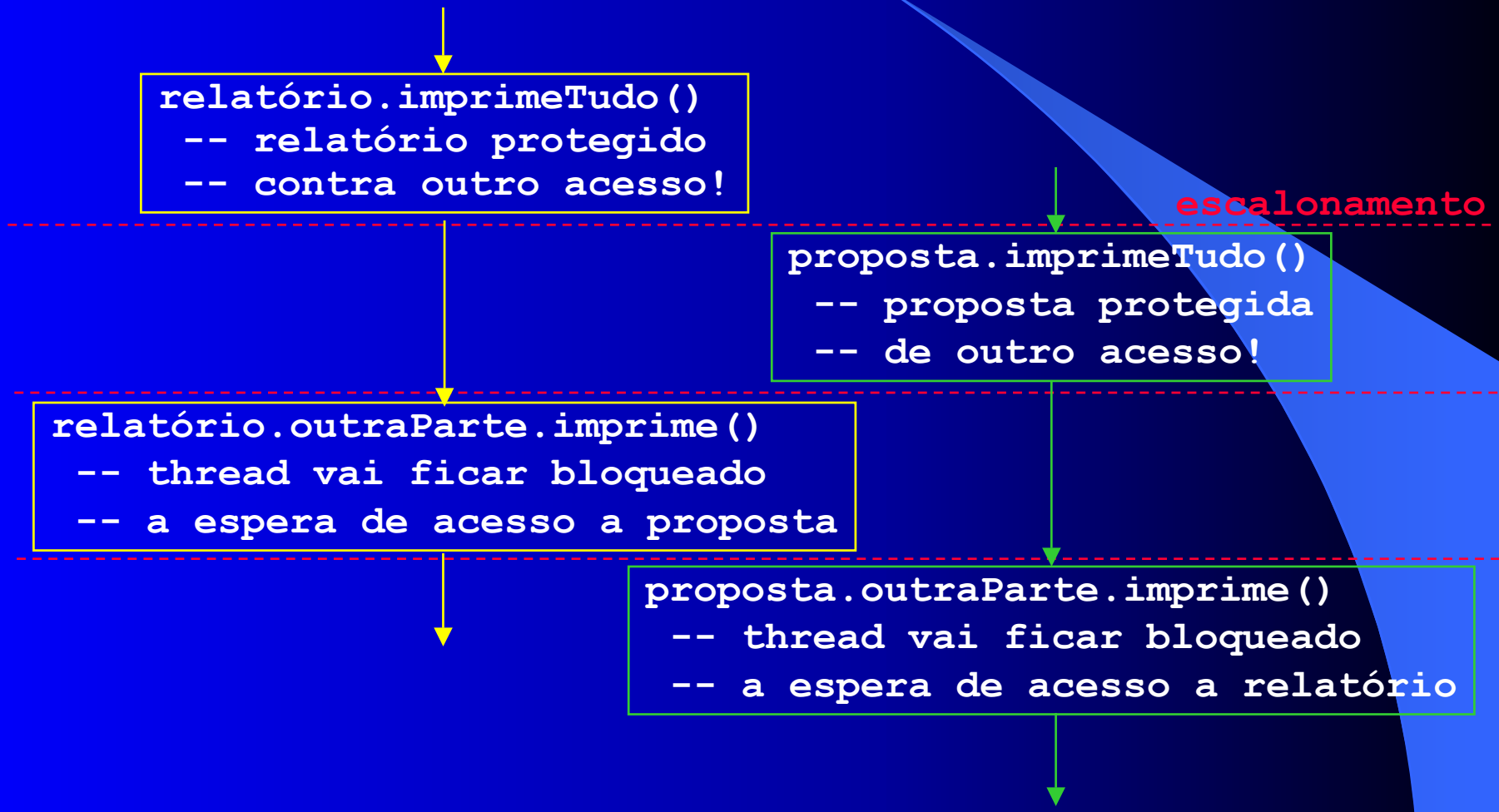
Continuando o exemplo

- Supondo que dois objetos **Documento** fazem referências um ao outro



- A execução de dois *threads* que imprimam estes documentos pode resultar em *deadlock*

Condições de corrida

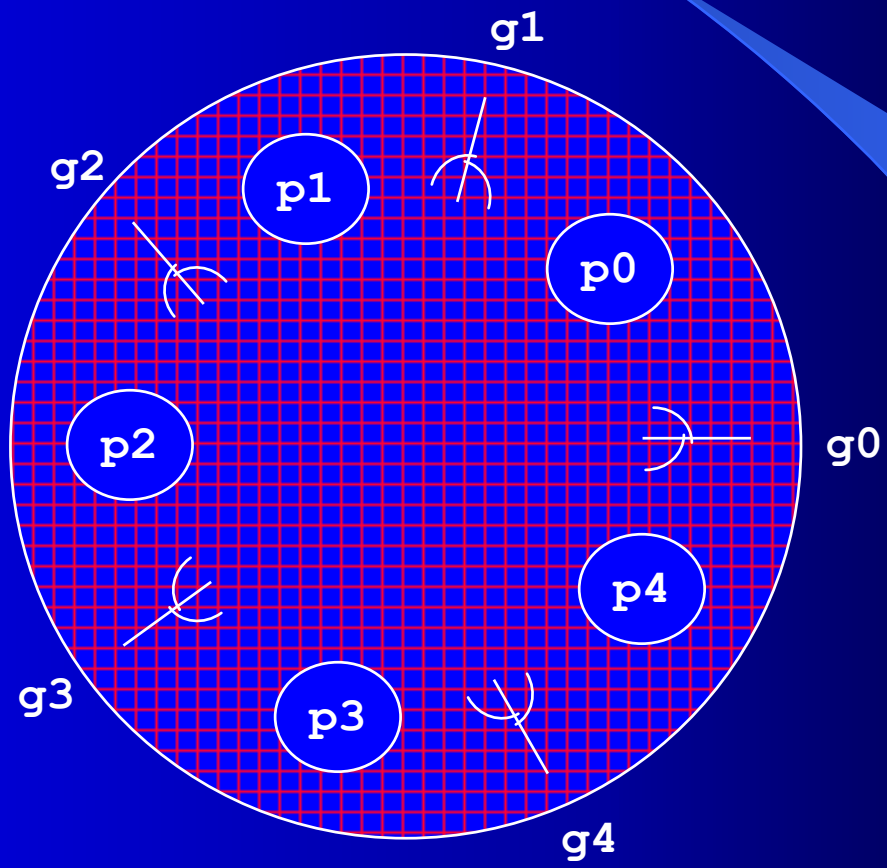


Starvation

- Se, ao chegar à ponte, há um carro já atravessando na direção desejada, o carro que chegou atravessa também
- Se há alguém vindo em direção oposta, o carro que chegou espera até que este tenha acabado de cruzar
- Pode nunca chegar a sua vez

Problemas Clássicos

Problema dos filósofos

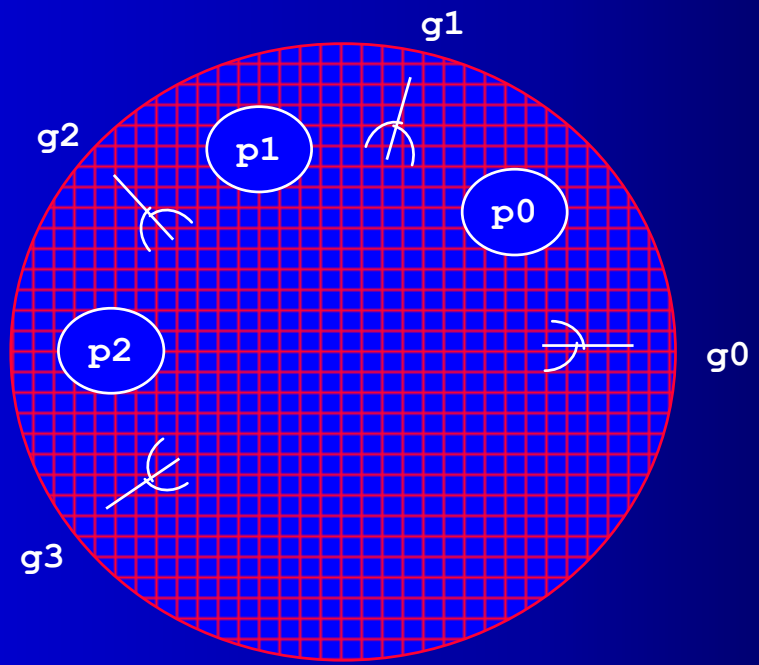


Filósofos - solução 1

```
class ContGarfos {
  private boolean[] livre = {true,true,true,true,true};
  ...
  public synchronized void pegaGarfos (int eu) {
    while (!(livre[esq(eu)] && livre[dir(eu)]))
      wait(); // omitindo tratamento de exceção!
    livre[esq(eu)] = false; livre[dir(eu)] = false;
  }
  public synchronized void liberaGarfos (int eu) {
    livre[esq(eu)] = true; livre[dir(eu)] = true;
    notifyAll();
  }
}
class Filosofo extends Thread {
  ContGarfos mordomo;
  ...
  public void run() {
    while (true) {
      // pensa
      mordomo.pegaGarfos(meuId);
      // come
      mordomo.liberaGarfos(meuId);
    }
  }
}
```

Problema

- Os filósofos 0 e 2 podem ficar em alternância e nunca deixar o filósofo 1 comer: *starvation*

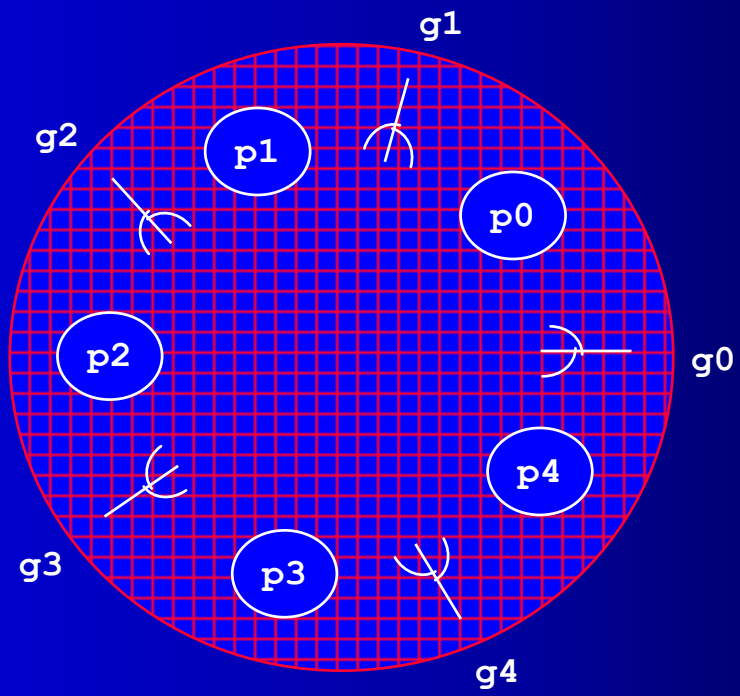


Filósofos - solução 2

```
class ContGarfos {
  private boolean[] livre = {true,true,true,true,true};
  ...
  public synchronized void pegaGarfos (int eu) {
    while (!livre[esq(eu)]) wait();
    livre[esq(eu)] = false; // SEGURA GARFO!
    while (!livre[dir(eu)]) wait();
    livre[dir(eu)] = false;
  }
  public synchronized void liberaGarfos (int eu) {
    livre[esq(eu)] = true; livre[dir(eu)] = true;
    notifyAll();
  }
}
class Filosofo extends Thread {
  ContGarfos mordomo;
  ...
  public void run() {
    while (true) {
      // pensa
      mordomo.pegaGarfos(meuId);
      // come
      mordomo.liberaGarfos(meuId);
    }
  }
}
```

Problema

- Cada filósofo pode ficar bloqueado com um garfo na mão: *deadlock*



Filósofos - solução 3

```
class ContGarfos {
  private boolean[] livre = {true,true,true,true,true};
  ...
  public synchronized void pegaGarfos (int eu) {
    int menor_garfo = Math.min(esq(eu),dir(eu));
    int maior_garfo = Math.max(esq(eu),dir(eu));
    while (!livre[menor_garfo]) wait();
    livre[menor_garfo] = false; // SEGURA GARFO!
    while (!livre[maior_garfo]) wait();
    livre[maior_garfo] = false;
  }
  public synchronized void liberaGarfos (int eu) {
    livre[Math.min(esq(eu),dir(eu))] = true;
    livre[Math.max(esq(eu),dir(eu))] = true;
    notifyAll();
  }
}
class Filosofo extends Thread {
  ContGarfos mordomo;
  ...
  public void run() {
    while (true) {
      // pensa
      mordomo.pegaGarfos(meuId);
      // come
      mordomo.liberaGarfos(meuId);
    }
  }
}
```

Técnica para evitar *deadlock*

- Cada filósofo pede o garfo da esquerda e depois o da direita, com exceção do primeiro, que pede ao contrário
- Exemplo de técnica mais geral:
  - ordena-se os recursos pelos quais há espera
  - pedidos são feitos sempre em ordem crescente
- Nem sempre faz sentido ordenar recursos

Ordenação em Java

- A classe **Object** defini um método chamado **hashCode** que pode ser usado como base da ordenação em algumas situações

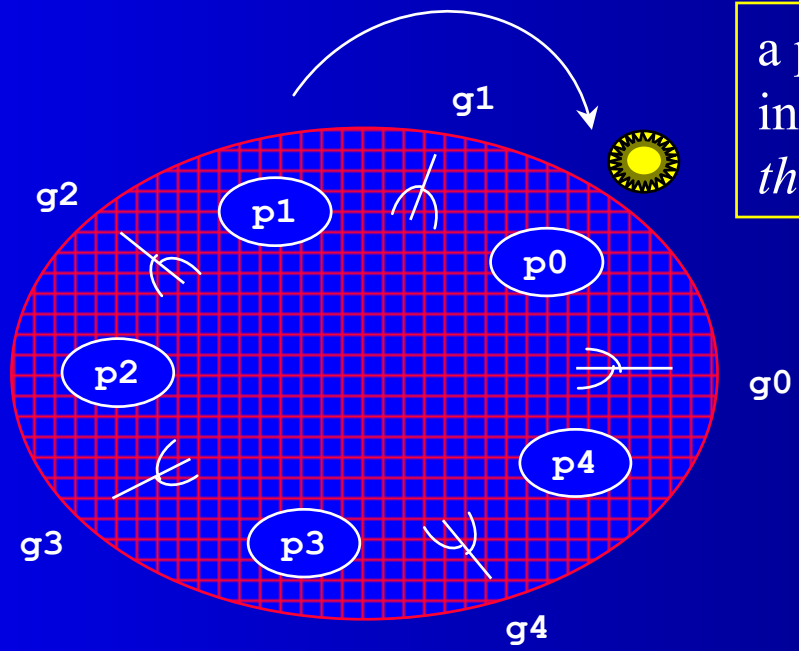
```
class Celula {
  private int valor;
  public synchronized int pegaValor () {
    return valor;
  }
  public synchronized void mudaValor (int v) {
    valor = v;
  }
  public void troca (Celula outra) {
    synchronized(this) {
      synchronized(outra) { // possibilidade de deadlock!
        int nv = outra.pegaValor();
        outra.mudaValor(valor); valor = nv;
      }
    }
  }
}
```

Exemplo de ordenação

```
class Celula {
  private int valor;
  public synchronized int pegaValor () {
    return valor;
  }
  public synchronized void mudaValor (int v) {
    valor = v;
  }
  public void troca (Celula outra) {
    Celula prim = this; Celula seg = outra;
    if (this.hashCode() > outra.hashCode()) {
      prim = outra; seg = this;
    }
    synchronized(prim) {
      synchronized(seg) {
        int nv = outra.pegaValor();
        outra.mudaValor(valor); valor = nv;
      }
    }
  }
}
```

Exemplo de restrição excessiva

- Uma forma comum de se resolver problemas de exclusão mútua é através do uso de um *token*



a presença do *token* indica que é a vez deste *thread* usar os recursos

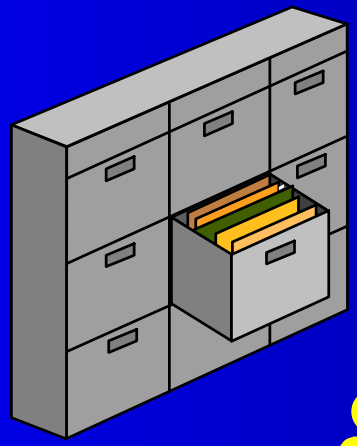
Resumindo restrição excessiva

- Não causa *deadlock* nem *starvation* mas elimina chance de concorrência

Circulação de token

```
class ContGarfos { // sem exceções: simplicidade
  private boolean[] token = {true,false,false,false,false};
  ...
  public synchronized void pegaGarfos (int eu) {
    while (!token[eu]) wait();
  }
  public synchronized void liberaGarfos (int eu) {
    token[eu]=false; token[direita(eu)]=true;
    notifyAll();
  }
}
class Filosofo extends Thread {
  ContGarfos mordomo;
  ...
  public void run() {
    while (true) {
      // pensa
      mordomo.pegaGarfos(meuId);
      // come
      mordomo.liberaGarfos(meuId);
    }
  }
}
```

Problema dos Leitores e Escritores



Leitores

Escritores

Leitores e Escritores

- Vários leitores podem consultar uma base de dados simultaneamente
- Um escritor deve ter acesso exclusivo

Proposta de solução

- Supondo que:
  - um *thread* pode iniciar uma leitura desde que não haja nenhum *thread* escrevendo na base;
  - um *thread* só pode iniciar uma escrita se não houver ninguém lendo e ninguém escrevendo na base.
- Proposta: criar classe **ControlaLE** com métodos:
  - **iniciaLeitura**
  - **finalizaLeitura**
  - **iniciaEscrita**
  - **finalizaEscrita**

Leitores e Escritores - solução 1

```
class ControlaLE {
  private int lendo, escrevendo;
  ...
  public synchronized void iniciaLeitura () {
    while (escrevendo!=0) wait();
    lendo++;
  }
  public synchronized void finalizaLeitura () {
    lendo--;
    if (lendo==0) notifyAll();
    // poderia notificar sem teste mas o preço seria alto!
  }
  public synchronized void iniciaEscrita () {
    while (lendo!=0 || escrevendo!=0) wait();
    escrevendo++;
  }
  public synchronized void finalizaEscrita () {
    escrevendo--;
    notifyAll();
  }
}
```

Alternativa

- Supondo que se deseje dar maior prioridade a escritores:
  - como programar a classe **ControlaLE** para um leitor não começar a ler se já houver algum escritor esperando?

Leitores e Escritores - solução 2

```
class ControlaLE {
  private int lendo, escrevendo, querendo_escrever;
  ...
  public synchronized void iniciaLeitura () {
    while (escrevendo!=0 || querendo_escrever!=0) wait();
    lendo++;
  }
  public synchronized void finalizaLeitura () {
    lendo--;
    if (lendo==0) notifyAll();
  }
  public synchronized void iniciaEscrita () {
    querendo_escrever++;
    while (lendo!=0 || escrevendo!=0) wait();
    querendo_escrever--;
    escrevendo++;
  }
  public synchronized void finalizaEscrita () {
    escrevendo--;
    notifyAll();
  }
}
```

Mecanismos para Construção e Gerência de *Threads* em Java

Outra forma de criar *threads*

- A obrigação de estender a classe **Thread** limita a implementação de uma classe, já que só há herança simples em Java
- A interface **Runnable** abstrai o conceito de algo "executável" e só define o método **run**
- Um objeto do tipo **Runnable** pode ser passado como argumento para um construtor da classe **Thread**

Exemplo de **Runnable**

```
class Conta implements Runnable {
  public void run() {
    int n = 0;
    while (true)
      System.out.println(n++);
  }
}

(new Thread(new Conta())).start
```

Interrompendo a execução de um *thread*

- Uma forma de se interromper a execução de um *thread* é "sugerir" que ele páre e esperar que ele próprio proceda com a interrupção
- Esse procedimento pode ser efetuado com o auxílio dos métodos **interrupt**, **isInterrupted** e **interrupted**

Interrompendo um *thread*

- Método **interrupt**
  - registra que o *thread* foi interrompido
  - *não* causa uma interrupção propriamente dita da execução do *thread*
  - interrompe qualquer chamada a **sleep**, **wait** ou **join**
- Método **isInterrupted**
  - indica se o *thread* foi interrompido
- Método **interrupted**
  - indica se o *thread* foi interrompido
  - registra que o *thread* não está mais interrompido

Exemplo de interrupção

```
class Trabalhador extends Thread {
  public void run() {
    while (!isInterrupted()) {
      ... // Trabalha um pouco
    }
  }
}

public void usuárioMandouParar() {
  if (confirma("Pára mesmo?")) // Confirma?
    trabalhador.interrupt(); // Pede para parar.
}
```

Métodos descontinuados

- **`public final void stop()`**
  - termina a execução
  - libera os monitores
    - *pode levar a um estado inconsistente*
- **`public final void suspend()`**
  - suspende a execução
  - não libera os monitores
    - *pode levar a um deadlock*
- **`public final void resume()`**
  - retorna a execução suspendida

Grupos de *threads*

- Normalmente, um novo *thread* é colocado no mesmo grupo do *thread* que o cria
- Construtores específicos permitem colocar o novo *thread* em outro grupo:
  - **Thread(ThreadGroup, Runnable)**
  - **Thread(ThreadGroup, Runnable, String)**
  - **Thread(ThreadGroup, String)**
- O grupo não pode ser alterado depois que o *thread* é criado

Classe ThreadGroup

- Construtores:
  - **ThreadGroup(String)**
  - **ThreadGroup(ThreadGroup, String)**
- Alguns métodos:
  - **public final ThreadGroup getParent()**
  - **public final synchronized void destroy()**
  - **public int enumerate(Thread[], boolean)**
  - **public synchronized void setMaxPriority(int)**

Prioridades

- O método **setMaxPriority** pode ser usado para limitar a prioridade máxima que um *thread* ainda a ser criado dentro de um grupo pode ter
- Os *threads* já existentes não são afetados

```
public void protegePrioridade(thread thr) {
  ThreadGroup grupo = thr.getThreadGroup();
  thr.setPriority(Thread.MAX_PRIORITY);
  grupo.setMaxPriority(thr.getPriority()-1);
}
```

Mais sobre escalonamento

- Não há garantias sobre preempção de *threads* entre *threads* de mesma prioridade
  - necessidade de mexer em prioridades para garantir atendimento
- Pode-se tentar construir um *thread*, com prioridade maior que os demais, com a função de garantir a preempção

```
public void run() {
  for (;;) {
    try { wait(fatiadetempo); }
    catch (InterruptedException e) {}
  }
}
```

Mesmo assim, nada se pode afirmar sobre quem vai ser escalonado!

Escalonamento Forçado

- Método **yield** interrompe o *thread* corrente para dar chance a um outro *thread* com prioridade igual ou maior
- Se houver apenas *threads* com prioridade inferior, o método **yield** retorna imediatamente
- Poderia ter sido usado no exemplo de acesso a saldo bancário, ao invés da chamada **sleep(10)**
- Não serviria se quiséssemos forçar um escalonamento entre *threads* de diferentes níveis de prioridade

Término de uma aplicação

- Uma aplicação termina quando todos os seus *threads* tiverem terminado
- O *thread* que executa o método **main** não possui nenhuma característica especial—ele é um *thread* como outro qualquer
  - se o método **main** não cria nenhum novo *thread*, a aplicação termina quando **main** retornar
  - se **main** criar um novo *thread* e depois retornar, a aplicação só terminará quando esse *thread* terminar

Não estamos considerando o término forçado, como, por exemplo, através de System.exit()

Tipos de *threads*: *User* × *Daemon*

- Um *daemon thread* é um *thread* cuja única função é prestar serviços a outros *threads* da aplicação
- Exemplos: redesenhar o *canvas*, imprimir documentos a partir de uma fila de impressão
- Um *user thread* é um *thread* principal, que executa tarefas e pode utilizar os serviços prestados pelos *daemons*
- A característica fundamental de um *daemon thread* é ser dispensável quando não houver ao menos um *user thread* executando

User × *Daemon*

- Um *thread* herda o seu *status* (*daemon* ou *user*) do *thread* que o criou
- Esse *status* pode ser alterado, antes que o *thread* comece a executar, através do método **setDaemon**
- A JVM se encarrega de detectar a situação onde apenas *daemon threads* estão executando e, nesse caso, termina o programa

```
Thread clock = new Watch();   // Exibe um relógio da tela
clock.setDaemon(true);        // Esse thread é um daemon
clock.start();                // Começa a trabalhar
```

Esperando por um *thread*

- Podemos esperar pelo término da execução de um *thread* chamando seu método **join**
- Assim como **sleep** e **wait**, **join** pode ser interrompido pelo método **interrupt**, causando o lançamento de uma exceção do tipo **InterruptedException**
- É possível limitar o tempo de espera

Método **join**

```
public final synchronized void join()
   throws InterruptedException
public final synchronized void join(long millis)
   throws InterruptedException
public final synchronized void join(long millis,
   int nanos) throws InterruptedException
```

Exemplo de espera

```
CalcThread calc = new CalcThread();
calc.start(); // Começa a calcular
... // Faz alguma outra coisa
try {
  calc.join(); // Espera pelo término do cálculo
  System.out.println("Resultado: "+calc.getResult());
} catch (InterruptedException ex) {
  System.out.println("Algo errado...");
}
```