

Processos e Threads

Computação de Alto Desempenho

Agenda – Parte I

- Conceitos
- Programação Paralela/Concorrente
- Programas e Processos
- Thread em Java
- Problemas com Concorrência
- Interação entre Processos
- *Spin-lock*
- Eventos
- Semáforos
- Mensagens
- RPCs
- Monitor

Conceitos

- **Objetivo:**
 - Melhorar performance/desempenho
- **Processo** – programa em execução
 - *ready* → *running* → *blocked* → ...
- **Multi-programação:** executa dois ou mais processos simultaneamente
 - Compartilhando CPU e dispositivos E/S

Conceitos

- **Multi-acesso:** executa dois ou mais processos controlados a partir de terminais interativos individuais
 - Compartilhando a memória principal
 - Memória virtual (*swapping* e *paging*)
- **Multi-processador:** mais de uma CPU executando vários processos em paralelo
 - Compartilhando a memória principal

Conceitos

- **Sistemas Distribuídos:** computadores independentes que podem se comunicar
- **Arrays de Processadores (*Transputers*):**
 - número de elementos computacionais (100 ou mais) que operam simultaneamente em diferentes partes de uma mesma estrutura de dados
- **Máquinas de fluxo de dados (*pipelines*)**
- **Grids computacionais**

Conceitos

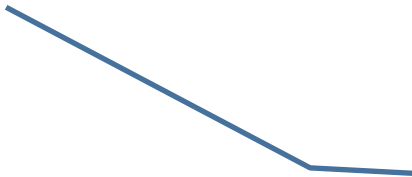
- **Programação estruturada** estabelece normas e estilos para programas seqüenciais
- **Programas seqüenciais** são mais simples devido às características:
 - Determinismo
 - Independência de velocidade de execução
 - Livre de *Deadlock*
 - Livre de Starvation (escalonamento desonesto)

Programas e Processos

- **Processo Seqüencial**
 - Conjunto ordenado de eventos, onde cada evento provoca mudança de estado em algum componente do sistema (Pascal)
- **Programa Seqüencial**
 - É um texto que especifica um processo seqüencial
- **Programa Paralelo/Concorrente**
 - Especifica mudanças de estado em dois ou mais processos seqüenciais

Programas e Processos

- **Processos concorrentes** não tem ordem de execução definida, executam simultaneamente ou concorrentemente
- **Pipeline** – permite que diferentes instruções sejam executadas ao mesmo tempo, porém em estágios diferentes (linha de montagem)



Exemplo (execução de 3 instruções):

fetch i3

decode i2

execute i1

Thread em Java

- Fluxo seqüencial de controle dentro do programa
- Uma thread é similar a um processo real (fluxo seqüencial em execução)
- Porém, uma thread é considerada um processo leve (*lightweight*), porque executa no contexto de um programa e usa os recursos alocados para este programa
- É possível criar processos em Java com `ProcessBuilder`

Thread em Java

- Como criar Threads?
 - Implementando a interface Runnable
 - Criando uma subclasse de Thread
- Se a subclasse deve estender outra classe, então é melhor implementar a interface Runnable

Thread em Java

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}

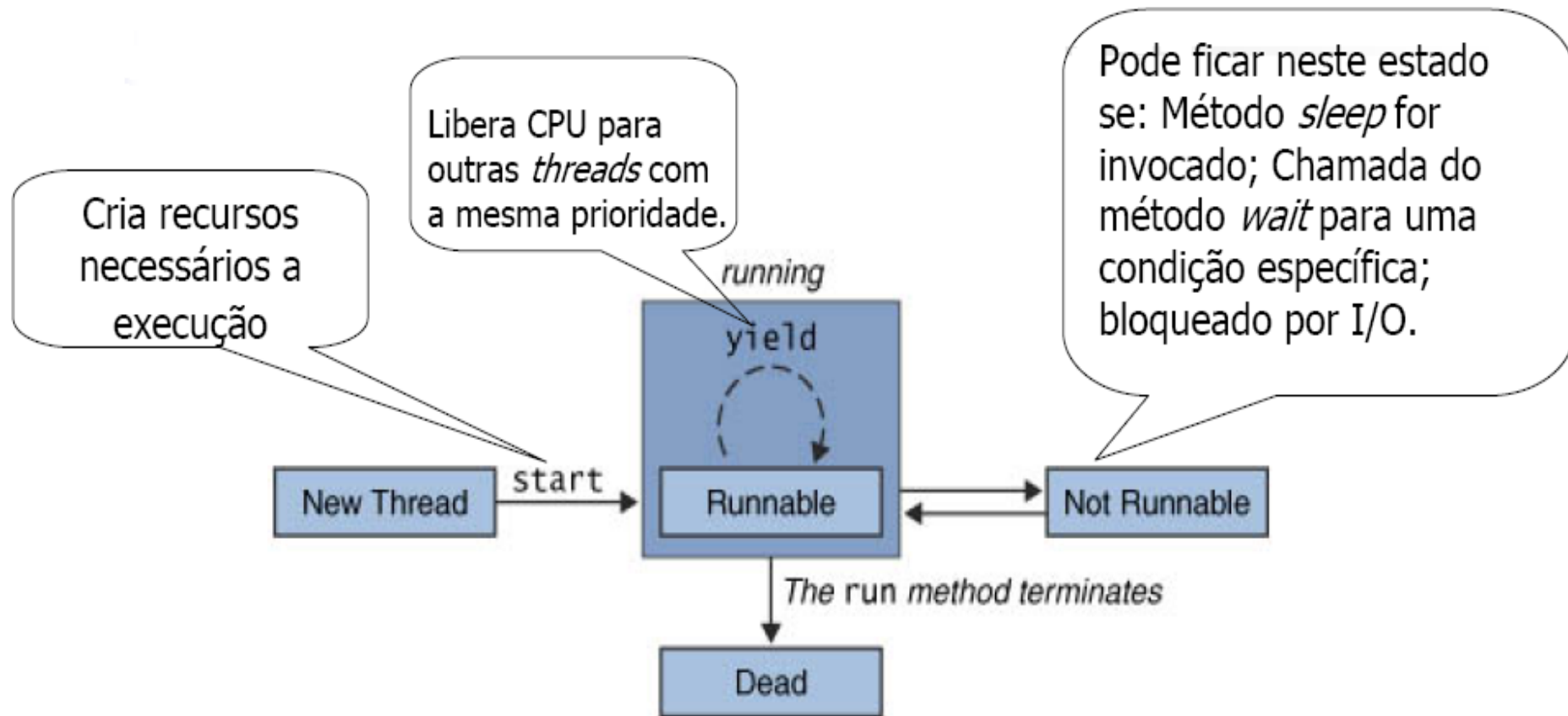
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Thread em Java



Criando uma Thread...

```
public class SimpleThread extends Thread {  
    public SimpleThread(String str) {  
        super(str);  
    }  
    public void run() {  
        for (int i = 0; i < 10; i++) {  
            System.out.println(i + " " + getName());  
            try {  
                sleep((long) (Math.random() * 1000));  
            } catch (InterruptedException e) {  
            }  
        }  
        System.out.println("DONE! " + getName());  
    }  
}
```

Problemas com Concorrência

- Não-determinismo
- Dependência de velocidade de execução (*race conditions*)
- *Deadlock*
- *Starvation*

Não-determinismo

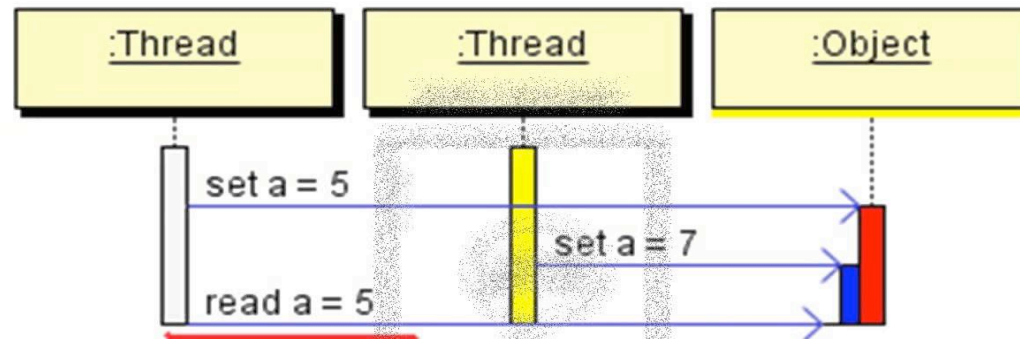
- Uma computação pode ser:
 - **Determinística** – é possível prever a seqüência de passos a ser seguida
 - **Não-determinística** – caso contrário
- **Exemplo:**
 - Considere os comandos C1 e C2. O comportamento depende da ordem em que forem executados
- Programas concorrentes/paralelos são, em geral, não-determinísticos

Dependência de Velocidade (*Rece Conditions*)

- Programas seqüenciais são independentes de velocidade
 - O mesmo programa executa corretamente em qualquer tipo de computador
- Programas concorrentes/paralelos podem ser dependentes de velocidade
 - O resultado de sua execução pode depender da velocidade de execução de seus processos seqüenciais componentes
- Em programas em tempo real, velocidades devem ser levadas em conta

Dependência de Velocidade (*Race Conditions*)

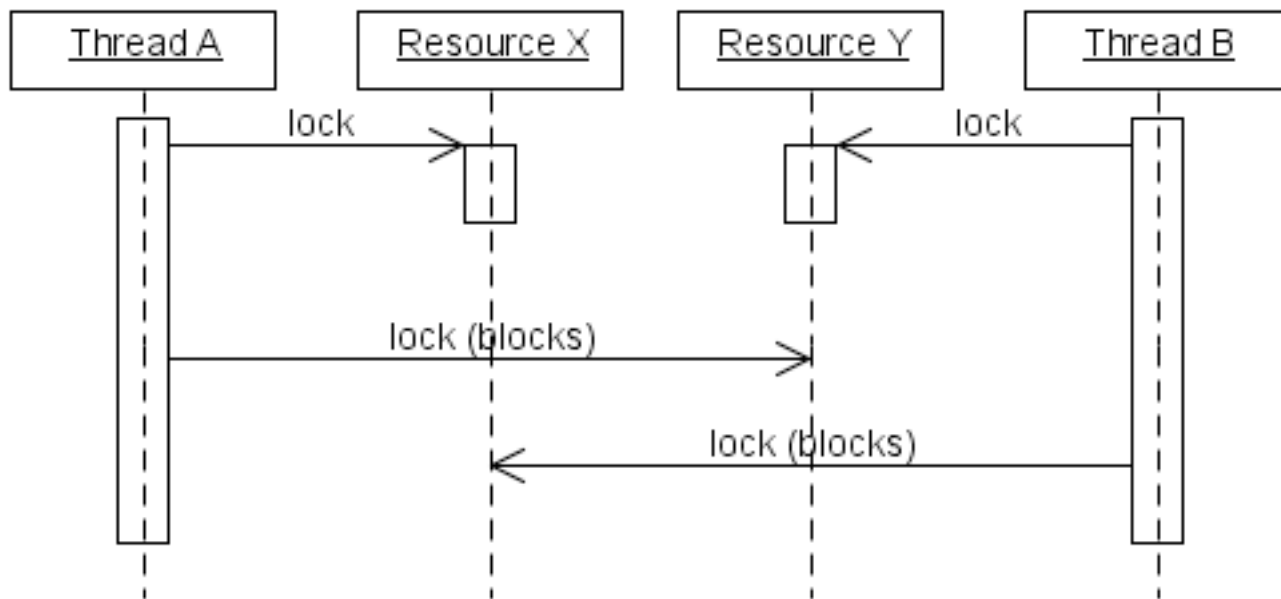
- A saída do programa é afetada pela ordem que as threads são alocadas na CPU
 - 2 threads modificam simultaneamente o mesmo objeto, então as duas threads “correm” (*race*) para armazenar seu valor



Race Condition:
Should be 7 not 5

Deadlock

- Situação onde um conjunto de processos estão impossibilitados de progredir porque eles dependem mutuamente do recurso alocado por outro.



Deadlock

- *Deadlock* ocorre se e somente se as seguintes condições ocorrerem simultaneamente:
 - **Exclusão mutua** – acesso exclusivo a recursos
 - **Espera e mantém** – processos mantêm recursos previamente alocados enquanto esperam por um novo recurso
 - **Não-preempção** – recursos de um processo não podem ser desalocados até que ele voluntariamente o renuncie
 - **Espera circular** – pode existir um ciclo de recursos e processos, onde cada processo espera por um recurso alocado pelo próximo processo no ciclo

Deadlock

- Existem várias abordagens para tratar o *deadlock*, onde cada uma possui uma série de técnicas/algoritmos associados:
 - Ignorar (Algoritmo do Avestruz)
 - Detectar e recuperar
 - Encerrando a execução de processos
 - Desalocando recurso (preempção)
 - Restaurando um estado anterior da execução ...
 - Prevenir, eliminando uma ou mais condições
 - Eliminar o espera e mantém
 - Eliminar a espera circular

Starvation

- Escalonamento é a alocação de recursos a processos
- Escalonamento honesto assegura que nenhum processo ficará esperando indefinidamente por um recurso devido a demanda de outros processos
- O termo *starvation* é usado para caracterizar um processo quando ocorre escalonamento desonesto
- *Starvation* ocorre quando um processo fica esperando indefinidamente para executar, por causa de um escalonamento desonesto/injusto
- Uma situação de *starvation* pode ocorrer quando um processo nunca é executado ("morre de fome"), pois processos de prioridade maior sempre o impedem de ser executado

Starvation

- É importante garantir que cada processo executará em um tempo finito
- Para isso é necessário:
 - Tratar o *deadlock*
 - Ter um escalonamento honesto (*fairly scheduled*)

Agenda – Parte II

- Interação entre Processos
- *Spin-lock*
- Eventos
- Semáforos
- Mensagens
- RPCs
- Monitor

Parte II

INTERAÇÃO ENTRE PROCESSOS

- Programas concorrentes diferem de programas seqüenciais porque os primeiros permitem a ocorrência de operações que causam interações entre processos
- C ; K (Seqüencial)
- C and K (Colateral)
- C || K (**Concorrente/Paralelo**)

INTERAÇÃO ENTRE PROCESSOS

- **Processos Independentes**

- **C** e **K** são comandos independentes se qualquer componente C_i de **C** pode ser processado em qualquer instante da execução de qualquer componente K_j de **K**
- **C**; **K** \equiv **K**; **C** \equiv **C** and **K** \equiv **K** || **C**
- Composição concorrente de processos independentes é determinística
- Se **C** e **K** são independentes, nenhum deles altera uma variável que é inspecionada pelo outro

INTERAÇÃO ENTRE PROCESSOS

- **Processos Competidores**

- **C** e **K** são comandos que competem se necessitam de acesso exclusivo a um mesmo recurso

- Dado que:

- **C** é a seqüência **C1**, **C2**, **C3**
- **K** é a seqüência **K1**, **K2**, **K3**
- **C1** e **K1** são independentes
- **C3** e **K3** são independentes
- **C2** e **K2** requisitam acesso exclusivo a um recurso **r**
- **C2** e **K2** não devem ocorrer simultaneamente nem no mesmo intervalo de tempo

INTERAÇÃO ENTRE PROCESSOS

- **C** e **K** são chamados de regiões críticas com respeito ao recurso **r**
- Podem ocorrer:
 - ...; **C2**; ...; **K2**;
 - ...; **K2**; ...; **C2**;
 - Nunca **K2** || **C2**
 - Em geral **K** || **C** é não determinístico
- Para garantir determinismo, é necessário que os processos se comuniquem

INTERAÇÃO ENTRE PROCESSOS

- **Processos Comunicantes**

- Dados dois comandos **C** e **K**, dizemos que existe comunicação de **C** para **K** se a ação **C2** deve preceder totalmente a ação **K2**, pois a primeira produz alguma informação que a segunda consome
- Aqui **C || K** tem o mesmo efeito que **C; K**

- Canal de processos: a saída de um processo serve de entrada para um próximo processo
 - Unix: **C1 | C2**
- Processos **C** e **K** se intercomunicam se existe comunicação em ambas as direções

Primitivas de Concorrência

- Criação e controle de processo

- *create*

- *load*

- *start*

- *suspend*

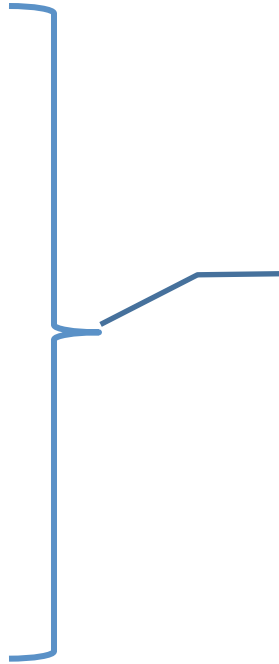
- *stop*

- *wait*

- *destroy*

- *fork* = *create* + *load* + *start*

- *join* = *wait* + *destroy*



Permite criar quaisquer sistema de processos concorrentes

Primitivas de Concorrência

- Outras primitivas são necessárias
- Tratamento de regiões críticas
 - *acquire(r)* – obter acesso ao recurso ***r***
 - *relinquish (r)* – liberar o acesso ao recurso ***r***
- Se ***r*** está alocado
 - *acquire(r)* bloqueia o processo que a executa
 - Quando ocorre o *relinquish (r)*, ***r*** é liberado e logo realocado para um dos processos que foi bloqueado por *acquire(r)*
- Comunicação entre processos
 - *transmite(c)*
 - *receive(c)*

Concorrência

- Como prover exclusão mútua?
 - Exclusão mútua (mutex) é uma técnica usada para evitar que dois ou mais processos ou threads acessem regiões críticas simultaneamente.
- **Região crítica** – trecho de código onde ocorre o acesso ao recurso
- Como gerenciar acesso a regiões críticas?
 - *Spin-locks*
 - Eventos
 - Semáforos
 - Mensagens
 - RPC

Spin-lock

- ***Spin-lock*** é um loop de espera ocupada
 - Determina que um processo espere por um acesso (exclusivo) a um recurso testando repetidas vezes um *flag* que indica se o recurso está livre ou não

Spin-lock

- Uma possível implementação:
 - Assume-se a existência de dois processos 1 e 2, cada um executando um programa com a seguinte forma (**self** sendo 1 ou 2), com um padrão cíclico de acesso ao recurso protegido r
 - **repeat**
 - código não crítico para processo self;*
 - `acquire(r);`
 - seção crítica para processo self;*
 - `relinquish(r);`
 - exit** *when processo self estiver finalizado;*
 - until** *processo self ser terminado*
 - `acquire(r):`
 - while** `flag = outro loop null; end loop;`
 - `relinquish(r):`
 - `flag:= outro`
- Usar uma variável `flag`, inicializada com 1 ou 2, que indica quais dos dois processos tem permissão para entrar na sua seção crítica.

Eventos

- Um evento representa uma classe de mudanças de estado que deve ser comunicada entre processos
 - *wait(e)* – processo é bloqueado até que ocorra um evento na classe de eventos identificada por **e**
 - *signal(e)* – acorda todos os processos que estão esperando pela ocorrência do evento **e**
- Desvantagens:
 - Operações *wait* e *signal* não são comutativas
 - Todos os processos são acordados (não há transmissão seletiva)
 - Não são úteis para exclusão mútua

SEMÁFORO

- Acesso Limitado, Exclusão Mútua e Sincronização
- Um semáforo é uma variável de um tipo abstrato que só pode ser acessada apenas por meio de três operações:
 - *sema-initialize(s, n)* – inicialização
 - *sema-wait(s)* – obtém acesso
 - *sema-signal(s)* – libera acesso

SEMÁFORO

- Estas operações são definidas em termos de seus efeitos sobre 3 valores inteiros associados com s :
 - $initial(s)$ – é o valor determinado por n na operação $sema-initialize(s, n)$ e que determina o número de processos permitidos
 - $wait(s)$ – é o número de operações $sema-wait(s)$ completadas
 - $signals(s)$ – é o número de operações $sema-signal(s)$ completadas
 - **$0 \leq wait(s) \leq signals(s) + initial(s)$**

SEMÁFORO

```
sema-wait(s) {  
    if( (signal(s)+initial(s))= wait(s) )  
        bloqueia o processo;  
    incremente wait(s);  
}
```

```
sema-signal(s) {  
    incremente signal(s);  
    if existe processo bloqueado  
        Escolha um para ser liberado  
}
```

SEMÁFORO

- `sema-initialize(s, 2)` -
`wait(s)=signal(s)=0`
- `P1 -> sema-wait(s)` - `wait(s)=1`
- `P2 -> sema-wait(s)` - `wait(s)=2`
- `P3 -> sema-wait(s)` - `P3 bloqueado`
- `P2 -> sema-signal(s)` - `signal(s)=1`,
`P3 continua, wait(s)=3`
- `P1 -> sema-signal(s)` - `signal(s)=2`

SEMÁFORO

- Dado que:
 - $wait(s) = initial(s) + signal(s)$
 - Se um processo executa *sema-wait(s)*, então ficará bloqueado até que outro processo complete uma invocação *sema-signal(s)*, incrementando o valor de *signal(s)*
 - Se muitos processos estão esperando em um mesmo semáforo, deve ser estabelecido um critério para escolher o processo a ser desbloqueado

SEMÁFOROS

- Operações de semáforo são comutativas;
- `sema-signal(e)` acorda apenas um processo (independente de velocidade)
- Podem ser usados para exclusão mútua e comunicação entre processos
 - Associar a cada recurso ***r*** um semáforo ***r-mutex***
 - Cada processo trata sua região crítica através das operações:
 - `sema-wait(r-mutex)` – *acquire(r)*
 - `sema-signal(r-mutex)` – *relinquish(r)*
 - `sema-initialize(r-mutex, 1)`
- Semáforos são mais apropriados para operações a nível de máquina

Mensagem

- Quando processos ocorrem em uma rede de computadores que não compartilham uma memória principal
 - *Spin-locks*, eventos e semáforos deixam de ser interessantes
- Comunicação se dá através de troca de mensagens entre processos
 - Através de um link
- O link pode suportar comunicação
 - Em uma única direção (*simplex*)
 - Em ambas direções (*half duplex*)
 - Em ambas direções concorrentemente (*full duplex*)

Mensagem

- Operações primitivas sobre links:
 - *connect* – conecta um processo ao link
 - *disconnect* – desconecta um processo de um link
 - *send* – envia uma mensagem sobre o link
 - *receive* – recebe uma mensagem de um link ou espera por sua chegada
 - *test* – verifica a chegada de uma mensagem no link
- A comunicação pode ser síncrona ou assíncrona

RPC

- Operações sobre o mesmo processo são invocadas através de chamadas de procedimento
- Operações sobre outros processos são invocadas através de envio de mensagens
- E se as responsabilidades de funções for mudada?
- *Remote Procedure Call* (RPC)
 - O ambiente de execução é quem determina onde um procedimento é provido

RPC

- Uma desvantagem do mecanismo de mensagens é que os programas ficam explicitamente divididos em processos
- Algumas operações são invocadas por chamadas de procedimento (dentro do mesmo processo) e outras por passagem de mensagens (por outros processos)
- Para manter a consistência, são usadas chamadas remotas de procedimento. (cliente-servidor)

Monitor

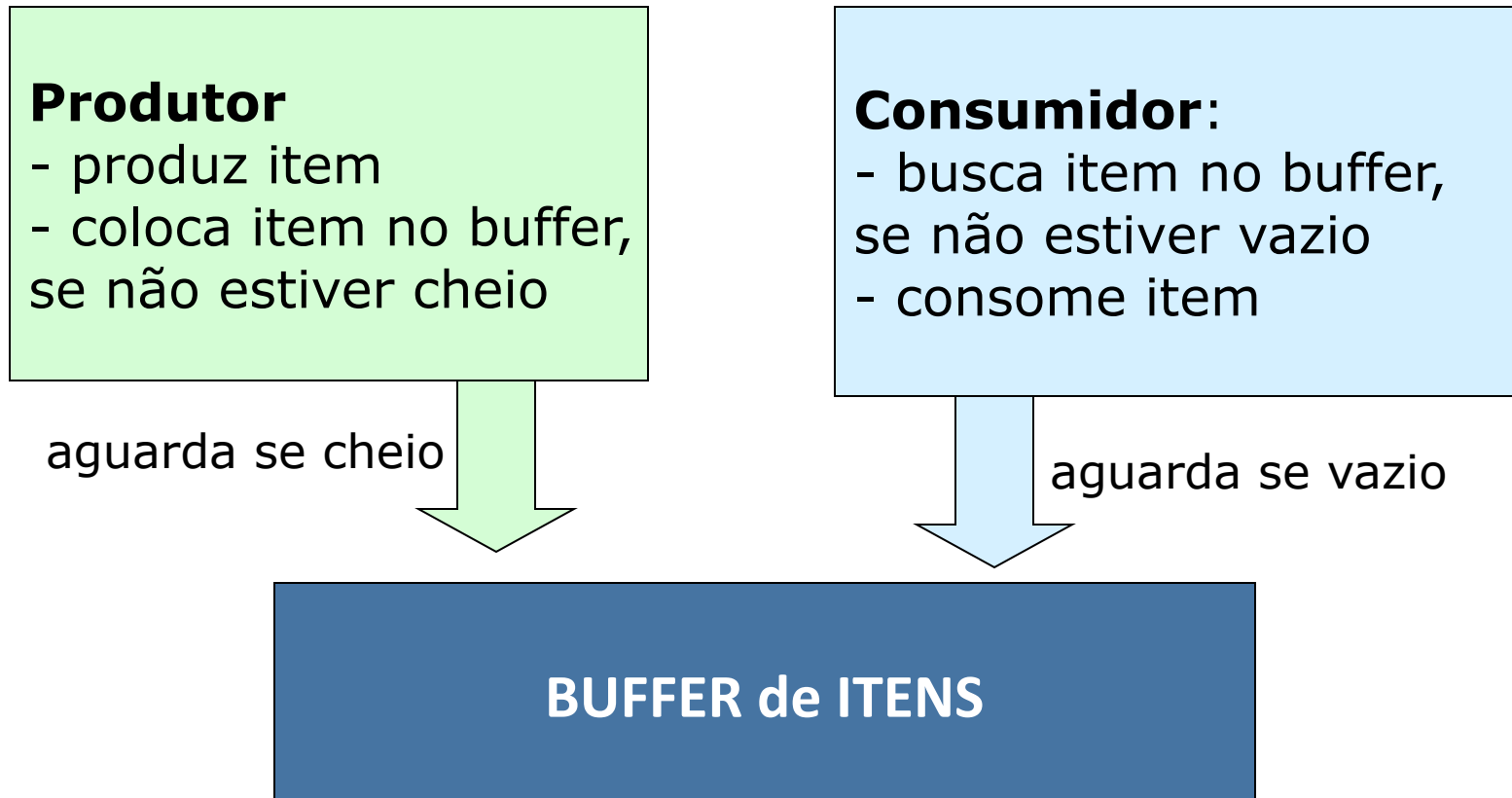
- Variáveis compartilhadas são encapsuladas em um módulo e tratadas como um objeto
- Um monitor combina encapsulamento, exclusão mútua e sincronização

Monitor

- A todo objeto em JAVA (variável de condição) que possui métodos síncronos (de acesso exclusivo - *synchronized*) é associado um monitor
- Seções críticas em programas JAVA são métodos síncronos
- A aquisição e liberação de um monitor é feita automaticamente e atomicamente pelo ambiente de execução de JAVA
- Exemplo em Java: Produtor-Consumidor

Monitor

Exemplo em Java: Produtor-Consumidor



Monitor - Exemplo em Java

Produtor-Consumidor

```
1 public class Consumidor extends Thread{
2     private Buffer pilha;
3     private int number;
4
5     public Consumidor(Buffer bf, int num){
6         this.pilha = bf;
7         this.number = num;
8     }
9
10    public void run(){
11        int valor = 0;
12        for(int i = 0; i < 10; i++){
13            valor = pilha.get();
14            System.out.println("Consumidor# " + this.number + " get: " + valor);
15        }
16    }
17 }
```

Monitor - Exemplo em Java

Produtor-Consumidor

```
1 public class Produtor extends Thread{
2     private Buffer pilha;
3     private int number;
4
5     public Produtor(Buffer bf, int num){
6         this.pilha = bf;
7         this.number = num;
8     }
9
10    public void run(){
11        for(int i = 0; i < 10; i++){
12            pilha.put(i);
13            System.out.println("Produtor# " + this.number + " put: " + i);
14            try{
15                sleep((long)(Math.random() * 1000));
16            } catch(InterruptedException e) {}
17        }
18    }
19 }
```

Monitor - Exemplo em Java

Produtor-Consumidor

```
1 public class ProdutorConsumidor {  
2  
3 public static void main(String[] args){  
4     Buffer bf = new Buffer();  
5     Produtor p1 = new Produtor(bf, 1);  
6     Consumidor c1 = new Consumidor(bf, 1);  
7     p1.start();  
8     c1.start();  
9 }  
10  
11 }
```

Monitor - Exemplo em Java

Produtor-Consumidor

Exclusão mutua é garantida?

Para cada *buffer* é criado um *lock (monitor)* tal que apenas um método qualificado com *synchronized* pode executar por vez.

Produtor e consumidor estão sincronizados?

O que acontece com *get se o produtor* não tiver colocado nenhum valor em *conteudo*?

O que acontece se o produtor chama *put antes* do consumidor coletar o valor?

```
1 public class Buffer {
2     int conteudo;
3
4     /**
5      * true- conteudo recebeu um valor
6      * false - o valor de contents foi coletado
7      */
8     boolean disponivel = false;
9
10    public synchronized int get(){
11        if(disponivel){
12            disponivel = false;
13        }
14        return conteudo;
15    }
16
17    public synchronized void put(int valor){
18        if(!disponivel){
19            disponivel = true;
20            conteudo = valor;
21        }
22    }
23
24 }
```

Monitor

Exemplo em Java: Produtor-Consumidor

- Sincronização: *wait e notify*
 - *Consumidor* deve esperar até que *Produtor* disponibilize um valor em *Buffer* e *Produtor* deve notificar *Consumidor* quanto isto ocorrer
 - Da mesma forma, *Produtor* deve esperar até que *Consumidor* acesse o valor antes de substituí-lo por um outro

Dúvidas?



EXERCÍCIOS

Exercício 1

- Defina a classe `Contador` como uma subclasse de `Thread`, que imprime números de 0 a 10. Crie a classe `TesteContador` que deve definir o método `main` que cria e inicia a execução do *thread* `Contador`. Teste o resultado executando a classe `TesteContador`.
- Altere as classes `Contador` e `TesteContador` de modo que a classe `Contador` seja definida como uma implementação da interface `Runnable`. Teste o resultado.
- Agora altere o método `main` da classe `TesteContador` para criar dois ou mais *threads* `Contador` e inicialize a execução dos mesmos.

Dicas:

- Na última parte do exercício, onde é pedido para alterar o método `main` da classe `TesteContador`, você deve inicialmente criar os *threads* e em seguida inicializá-los, de modo a obter um melhor resultado na visualização da execução concorrente.
- Como o *loop* do exemplo é apenas de 1 a 10 pode ser que não seja visualizado um *interleaving* entre as execuções dos *threads*. Caso isto ocorra aumente o tamanho do *loop*.

Exercício 2

- Defina uma classe `Mailbox` que tem um atributo `message` do tipo `String`. A classe `Mailbox` deve ter dois métodos: `storeMessage` e `retrieveMessage`. O método `storeMessage` recebe um `String` e, se o mesmo tiver sido consumido (`message == null`), armazena no atributo `message` do `Mailbox`. Caso contrário, quem chamou o método `storeMessage` deve esperar até que alguém consuma a mensagem (chamando o método `retrieveMessage`). De forma similar, o método `retrieveMessage` retorna o valor da mensagem, caso ela tenha sido produzida/armazenada (`message != null`). Caso contrário quem chamou o método deve esperar até que alguém produza uma mensagem (chamando o método `storeMessage`).
- Crie a classe `Producer`, que é um *thread* e deve ter um atributo do tipo `Mailbox` e no seu método `run` deve ser definido um *loop* que executa o método `storeMessage` do `Mailbox`, armazenando mensagens no `Mailbox`.
- Defina uma classe `Consumer`, que também é um *thread*, e que deve consumir mensagens (chamando o método `retrieveMessage`) escritas em no seu atributo do tipo `Mailbox`.
- Crie uma classe de teste com um método `main` que cria um `Producer` e um `Consumer` que compartilham o mesmo `Mailbox` e iniciam a execução.
- Altere a classe de teste para criar mais de um produtor e/ou consumidor para o mesmo `Mailbox`.

Dicas:

- Utilize os métodos `wait` e `notifyAll` para implementar os métodos da classe `Mailbox`.
- Na última parte do exercício, onde é pedido para alterar a classe de teste, você deve adicionar um atributo de identificação (`String`) nas classes `Producer` e `Consumer`, de modo a permitir que sejam identificados tanto que está consumindo uma mensagem, quanto que esta produzindo a mesma. Para tal concatene o identificador dos objetos a mensagem a ser enviada (no caso do produtor) ou a mensagem a ser impressa (no caso do consumidor).