FACAMP
FACULDADES DE CAMPINAS

Linguagens de Programação

Aula 9

Multithreading e concorrência

2º semestre de 2019

Prof José Martins Junior

Motivação

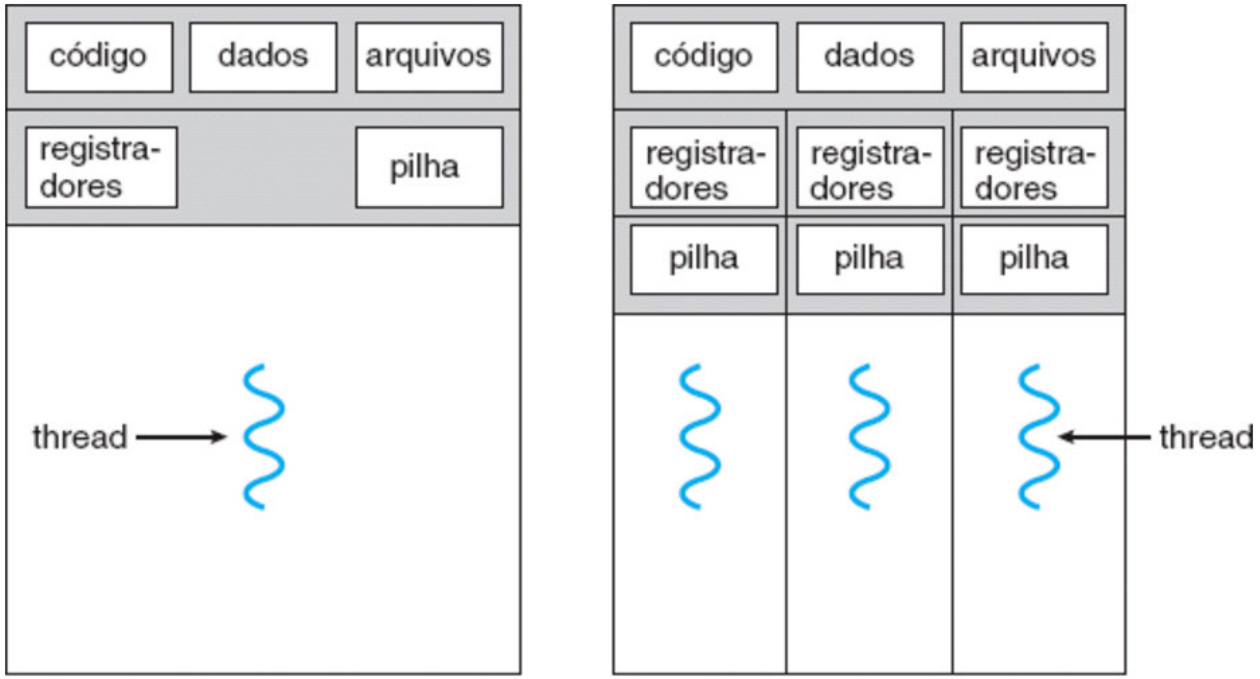
- Muitas aplicações hoje são multithreaded
  - Exemplo: um browser deve atender “simultaneamente” o usuário, enquanto processa solicitações de partes de uma página e as exibe
- Uma aplicação pode interagir com diversos sistemas
  - Vários laços de repetição com diferentes requisitos temporais
- O custo de um processo é mais pesado que de um thread
  - Threads também são chamados de light-weight process
- Threads oferecem facilidades no compartilhamento de dados
- Atualmente, os kernels também são multithreaded
  - Para tirarem vantagens de arquiteturas com múltiplos processadores ou núcleos

Benefícios do uso de threads

- Capacidade de resposta
  - Permite continuar a execução se parte do processo é bloqueada
  - Isso é muito importante para interfaces de usuários
- Compartilhamento de recursos
  - Threads compartilham recursos do processo original
  - Muito mais fácil que memória compartilhada ou troca de mensagens
- Economia
  - Geralmente, têm custo menor que a criação de um processo
  - O chaveamento de thread causa menos sobrecarga que o de contexto
- Escalabilidade
  - Pode obter vantagem do paralelismo em arquiteturas multiprocessadas

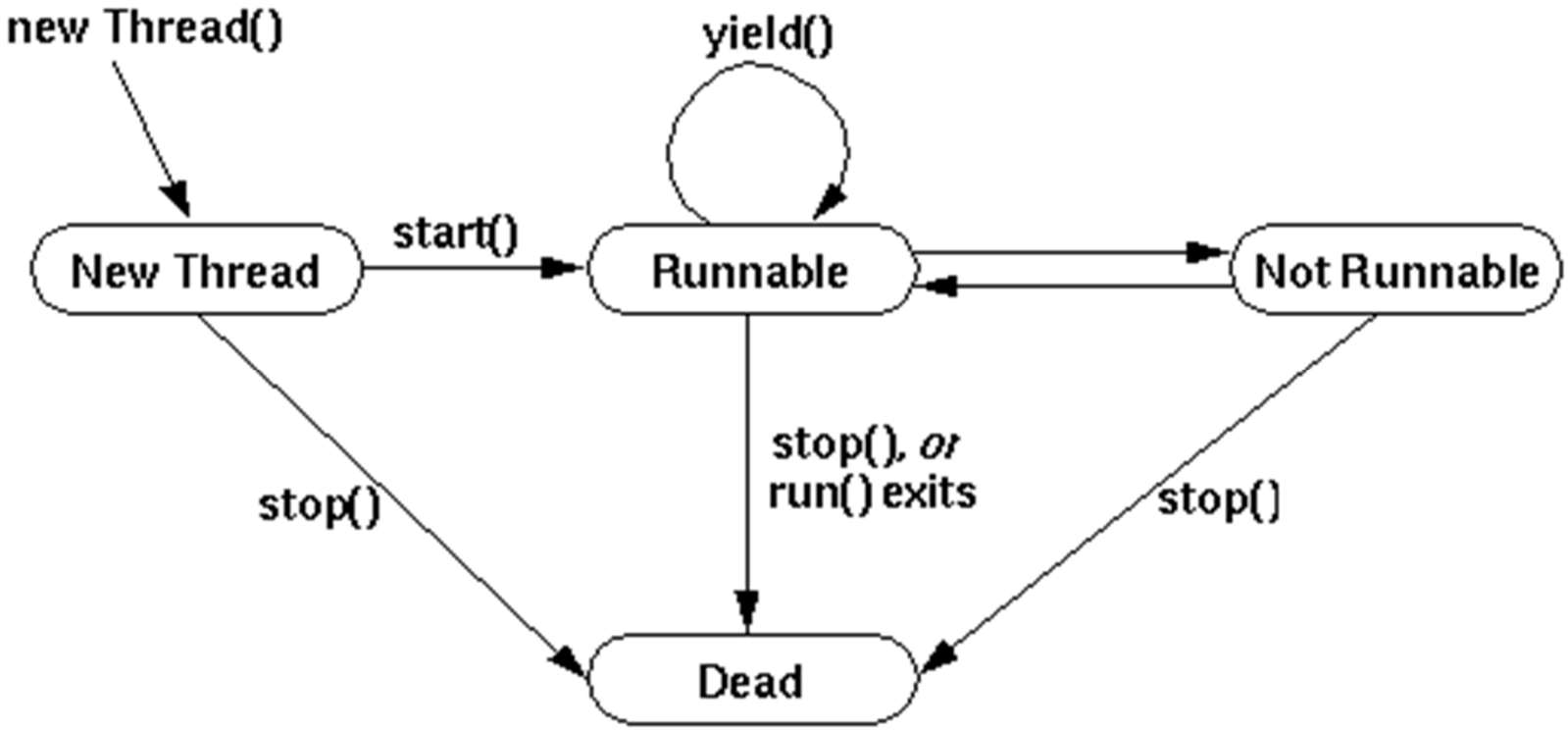
Um X vários threads

- Threads de um processo compartilham
  - Áreas de código (text section) e de dados
  - Descritores de arquivos abertos
- Cada thread possui seus próprios
  - Registradores, contador de programa e pilha de dados temporários



Java threads

- Threads em Java são gerenciados pela JVM
  - São geralmente implementados, de acordo com o modelo disponível no SO nativo
- Principais estados dos threads Java e métodos de transição



New Thread

- Criação do thread através do construtor Thread()

```
class MyThreadClass extends Thread {
    //...
}
//...
MyThreadClass myThread = new MyThreadClass();
```

  - Neste estado, nenhum recurso do sistema foi alocado ainda
    - Para ativá-lo, chama-se `myThread.start()`
    - Para encerrá-lo, chama-se `myThread.stop()`
  - Se outros métodos forem chamados neste estado, resultará uma exceção IllegalThreadStateException

Runnable

- Estado de um thread logo após um start()
  - A máquina virtual aloca os recursos necessários
  - Em seguida, chama o método run(), de implementação obrigatória por quem implemente a interface Runnable
  - Dentro do método run(), implementa-se a lógica do thread

- O estado chama-se Runnable,
  - Pois indica que o thread está pronto para ser escalonado pelo sistema
  - Quando recebe uma quota de tempo para execução, ele passa para o estado Running

Not Runnable

- Indica que o thread está impedido de executar

- Quatro possíveis razões
  - Outro processo envia-lhe a mensagem suspend()
  - Outro processo envia-lhe a mensagem sleep()
  - O thread está bloqueado, esperando por I/O
  - O thread usa seu método wait() para esperar por uma variável de condição

Not Runnable (sleep)

- Colocar “myThread” para dormir por 10 segundos

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
      myThread.sleep(10000);
} catch (InterruptedException e){
}
```

Saindo de um “Not Runnable”

- Formas de sair do estado “Not Runnable”
  - Se o thread foi suspenso, aguarda que outro método envie-lhe a mensagem resume()
  - Se o thread foi posto para dormir, retornará para “Runnable” quando o tempo em milissegundos passar
  - Se o thread está bloqueado, a operação de I/O precisa ser completada
  - Se o thread está esperando por uma variável de condição, o outro método pode liberá-lo com notify() ou notifyAll()

Dead

- Um thread pode morrer
  - Naturalmente, quando o seu método run() termina
  - “Assassinado” pelo método stop()
- Este thread vai morrer quando o loop do run() acabar

```
Thread myThread = new MyThreadClass();
myThread.start();
public void run() {
        int i = 0;
        while (i < 100) {
                i++;
                System.out.println("i = " + i);
        }
}
```

Dead

- Neste exemplo, o thread é encerrado por stop() 10 segundos após a sua inicialização
  - A chamada Thread.sleep() põe o chamador para dormir

```
Thread myThread = new MyThreadClass();
myThread.start();
try {
      Thread.sleep(10000);
} catch (InterruptedException e){
}
myThread.stop();
```

Método yield()

- Libera a CPU para outros threads
  - Quando, por exemplo, for previsto o encerramento prematuro de uma execução ou lógica
  - Ou, quando deseja-se colocar o thread para dormir o tempo restante de sua quota
  - Pode ser chamado pelo próprio thread em execução

```
Thread.yield();
```

Thread em Java

- Em Java, threads são implementados como classes
- Duas formas de implementar threads
  - Estender a classe java.lang.Thread
    - A vantagem é que pode ser instanciada e possuirá métodos de Thread para sua operação (start, stop, entre outros), como mostra o exemplo da MyThreadClass nos slides anteriores
    - A desvantagem é que Java não permite múltipla herança e pode não ser possível estendê-la, se a classe já estender outra
  - Implementar a interface java.lang.Runnable
    - A vantagem é permitir que qualquer classe seja transformada em um Thread, implementando o método run()
    - A desvantagem é que a classe não possuirá os métodos de Thread e dependerá de uma instância de Thread para executar

Principais métodos de Thread

- start(): método inicializa um thread e chama o método run( )
- run(): método que implementa a funcionalidade de um thread e não pode ser chamado (só pelo start())
- sleep(long x): coloca o thread para dormir por x milissegundos
- join( ): coloca o chamador para esperar o término do thread chamado
- interrupt( ): método que interrompe a execução de um thread
- isInterrupted( ): testa se um thread está ou não interrompido
- setPriority(int p): define a prioridade de execução do thread
- getPriority( ): obtém a prioridade de execução do thread
- isAlive(): verifica se o método run() do thread terminou ou não

Prioridade de um Thread

- Determinada com um inteiro entre 0 e 10

```
0  Thread.MIN_PRIORITY
5  Thread.NORM_PRIORITY (padrão)
10 Thread.MAX_PRIORITY
```

- Se nada for mudado, um thread herda a mesma prioridade do thread que o criou
- Cuidado com a modificação das prioridades
  - Em alguns sistemas, os threads com prioridades maiores são sempre chamados, o que pode gerar starvation (um thread de prioridade menor nunca executar) e deadlock (que pode ser decorrente de um starvation de um processo produtor, por exemplo)

Exemplo com Thread

```
class MyThread extends Thread {
  public void run() {
    System.out.println("Running");
  }
}

class TestThread {
  public static void main(String[] args) {
    MyThread thread = new MyThread();
    thread.start();
  }
}
```

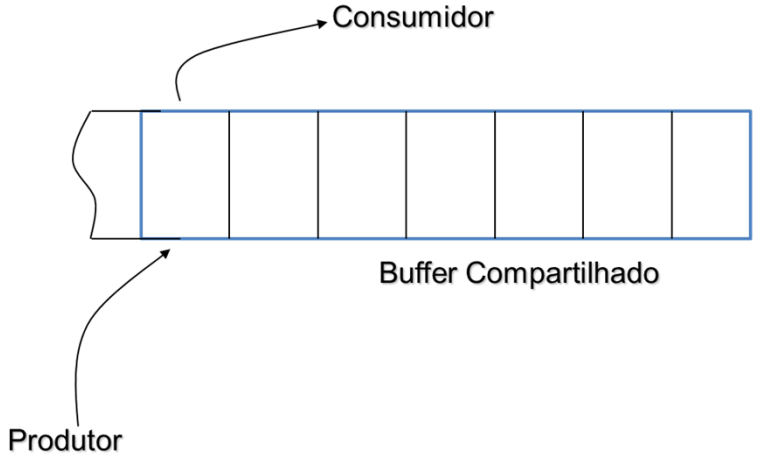
Exemplo com Runnable

```
class MyRunnable implements Runnable {
  public void run() {
    System.out.println("Running");
  }
}

class TestRunnable {
  public static void main(String[] args) {
    Thread thread = new Thread(new MyRunnable());
    thread.start();
  }
}
```

Produtor-consumidor

- Ilustração do problema
  - Um processo (thread) produtor e um produtor compartilham um buffer
    - O produtor insere no buffer e o consumidor retira
  - Eles compartilham dados comuns (array, variável contadora..)
    - Caso um modifique uma posição do array e não atualize o contador, o próximo sobrescreverá (ou lerá o mesmo valor): **inconsistência**
  - Além disso
    - Um produtor não pode inserir em um buffer cheio
    - E um consumidor não pode consumir de um buffer vazio



Seção crítica

- Considere n threads concorrentes
- Cada thread tem uma seção crítica em um trecho de código
  - Podem estar modificando variáveis comuns, atualizando uma tabela ou escrevendo em um arquivo
  - Quando uma thread executar a região crítica, nenhuma outra poderá

```
do {
        entry section
            critical section
        exit section
            remainder section
} while (true);
```

Mutex locks

- Mutex é abreviação de Mutual Exclusion
  - Portanto, visa garantir o primeiro requisito
- Define operações simples para bloqueio e desbloqueio
  - Para entrar na região crítica, o processo deve adquirir um bloqueio
    - Enquanto a variável de bloqueio não estiver disponível, realiza uma espera ocupada (busy wait)
  - Ao sair da região, o processo libera a variável de bloqueio

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;;
}
release() {
   available = true;
}
```

```
do {
   acquire lock
      critical section
   release lock
      remainder section
} while (true);
```

Semáforos

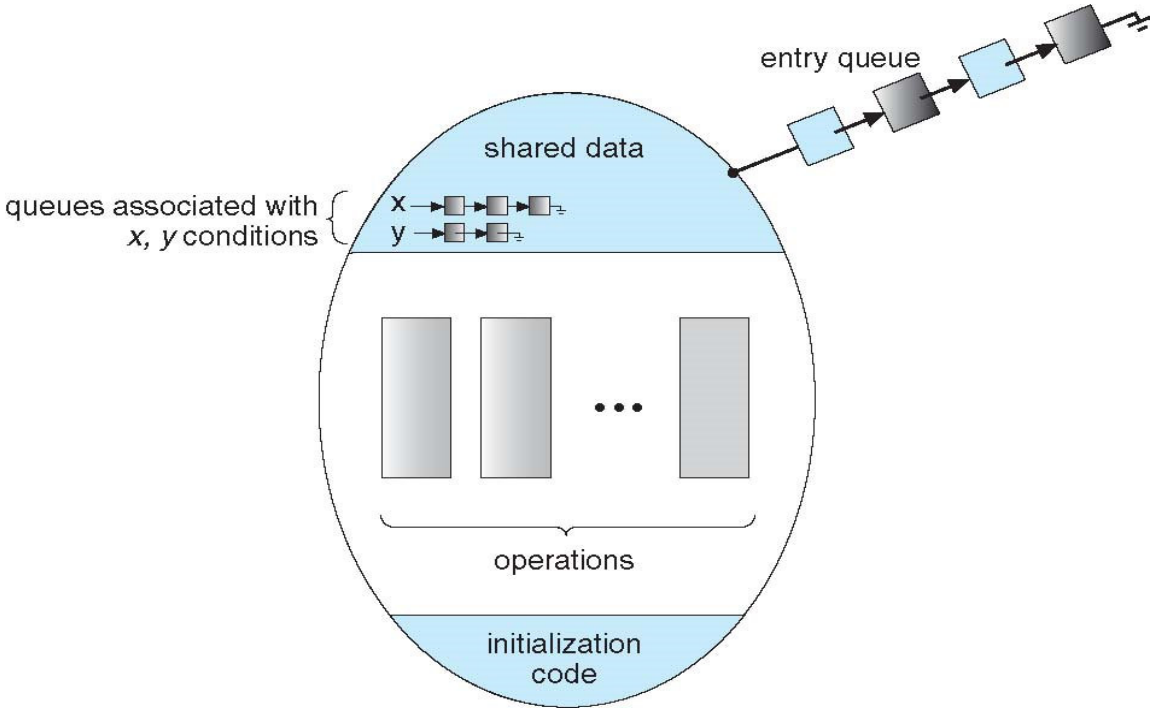
- Ferramenta de sincronização que não requer espera ocupada
- Um semáforo S é uma variável inteira
  - Indica quanto do recurso ainda está disponível
- Possui duas operações básicas
  - wait: testa S e, se negativa, entra em espera ocupada, senão decrementa S
  - signal: incrementa S

```
wait (S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal (S) {
    S++;
}
```

Monitores

- Abstrações de alto nível para mecanismos de sincronização
  - Fazem uso de estruturas internas de abstract data types (ADT) ou objetos, para encapsulamento de variáveis
  - Apenas um processo pode estar ativo no monitor por tempo
  - Podem definir variáveis de condição (bloqueio) para sincronização de operações distintas



Monitores em Java

- Todo objeto em Java possui um monitor
- Operador **synchronized**
  - Quando aplicado a um **método**, garante que só um thread/processo o acessará ao mesmo tempo, os demais são enfileirados
    - Métodos distintos possuem filas de acesso distintas

```
synchronized void incrementAll(int[] data){
  for(int i = 0; i < data.length; i++) data[i]++;
}
```

  - Se aplicado a um bloco, pode-se atribuir uma variável de bloqueio para identificar diferentes regiões críticas (recursos ou filas)

```
protected final Object lock = new Object();
void incrementAll(int[] data){
  synchronized(lock) {
    for(int i = 0; i < data.length; i++) data[i]++;
  }
}
```

Bloqueios entre threads em Java

- Objetos herdam métodos wait(), notify() e notifyAll() de Object
  - Tais métodos só podem ser chamados dentro de um contexto sincronizado
  - Ao chamar wait(), o thread é colocado para dormir e libera o monitor, permitindo que outros threads o obtenham e executem o código sincronizado
  - Ao chamar notify() ou notifyAll(), um ou mais threads são acordados para receber o monitor e voltar a obter sua quota de tempo de execução

Exemplo: Bank.java

```java
package aula9;

public class Bank {

    private String transName;
    private double total;
    private double transactionValue;

    void update(String transName, double amount) {
        this.transName = transName;
        this.transactionValue = amount;
        for (int i = 0; i < Math.abs(transactionValue); i++) {
            this.total += Math.signum(amount);
        }
        System.out.println(this.transName + " " + amount + " total = " + total);
    }
}
```

Exemplo: TransThread.java

```
1   package aula9;
2
4   public class TransThread extends Thread {
5
6       private Bank bank;
7       private String name;
8       private double amount;
9
10      public TransThread(Bank bank, String name, double amount) {
11          super(name);
12          this.bank = bank;
13          this.name = name;
14          this.amount = amount;
15      }
16
17      @Override
18      public void run() {
19          for (int i = 0; i < 100; i++) {
20              bank.update(name, amount);
21          }
22
23      }
24  }
```

Exemplos: DepositThread.java e WithdrawThread.java

```java
package aula9;

public class DepositThread extends TransThread {

    public DepositThread(Bank bank) {
        super(bank, "Deposit", +250);
    }

}
```

```java
package aula9;

public class WithdrawThread extends TransThread {

    public WithdrawThread(Bank bank) {
        super(bank, "Withdraw", -250);
    }

}
```

Exemplo: SynchronizationDemo.java

```
1   package aula9;
2
3   public class SynchronizationDemo {
4
5       public static void main(String[] args) {
6           Bank bank = new Bank();
7           TransThread tt1 = new DepositThread(bank);
8           TransThread tt2 = new WithdrawThread(bank);
9           tt1.start();
10          tt2.start();
11      }
12
13  }
```

Bibliografia

- SEBESTA, R. W. Conceitos de Linguagens de Programação. Porto Alegre: Bookman, 2011. 792p.
- SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. Fundamentos de Sistemas Operacionais. 9ª ed. Rio de Janeiro: LTC, 2015.