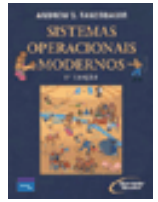


# Capítulo 2

# Processos e Threads

2.1 Processos
2.2 **Threads**
2.3 Comunicação interprocesso
2.4 Problemas clássicos de IPC
2.5 Escalonamento

- Programa Sequencial
  - Programa executado por apenas um processo
- Programa Concorrente
  - Programa executado por diversos processos que cooperam entre si para a realização de uma tarefa
    - Necessidade de interação para troca de informações

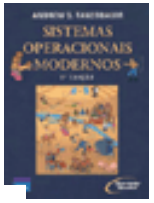
- Paralelismo Real
  - Execução simultânea de dois ou mais processos
  - Só ocorre em máquinas multiprocessadoras
  - Multiprocessamento físico (Dual Core e ou lógico (tecnologia Hyper-Threading da Intel)
- Paralelismo Aparente
  - Utiliza vários recursos, como compartilhamento de tempo de CPU entre vários processos, para simular simultaneidade
    - Dá impressão ao usuário de que os programas são executados ao mesmo tempo

# Programação Concorrente

- Composta por um conjunto de processos sequenciais que executam concorrentemente
- Processos disputam recursos comuns
  - variáveis, periféricos, etc.
- Processos cooperantes
  - Capaz de afetar, ou ser afetado, pela execução de outro processo

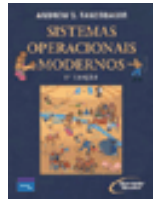
# Motivação para Programação Concorrente

- Aumento de desempenho
  - Permite exploração do paralelismo real disponível em máquinas multiprocessadoras
  - Sobreposição de operações de E/S com processamento
- Facilidade de desenvolvimento de aplicações que possuem um paralelismo intrínseco
  - Ex.: jogos, próprio Sistema Operacional



# Desvantagens

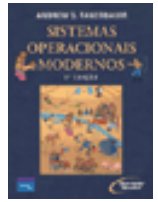
- Programação complexa
- Além dos de programação comuns, se adiconam os erros próprios do modelo
  - Erros associados às interações entre processos
- Difícil depuração



# Relação Produtor-Consumidor

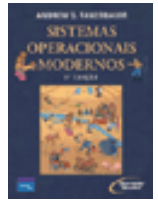
- Processo produtor produz um fluxo de dados consumido pelo processo consumidor
- Exemplo: programa que lê um arquivo, formata os dados e envia para a impressora
  - Programa Sequencial
  - Programa Concorrente

# Programa Sequencial



# Programa Sequencial

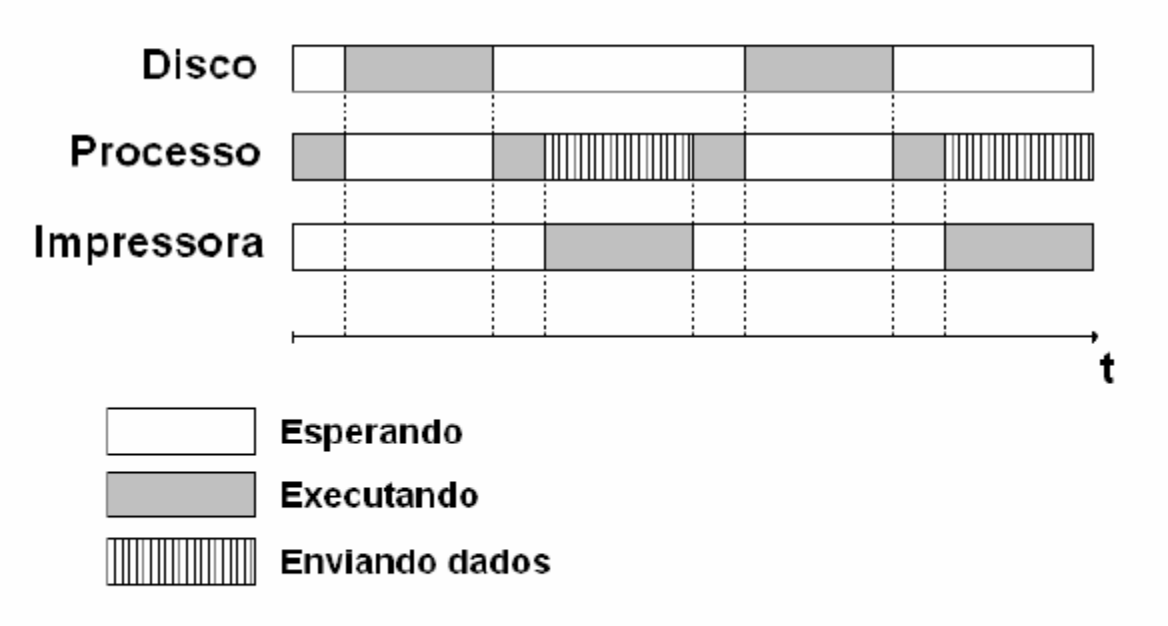
1. Processo envia um comando para leitura de arquivo e fica bloqueado
2. Disco é acionado para realizar operação de leitura
3. Concluída a leitura, processo realiza formatação e inicia a transferência dos dados para a impressora
4. Processo executa um laço no qual os dados são enviados para a impressora
   - Processo fica preso até o final da impressão

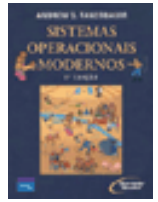


# Programa Sequencial

- Disco e impressora nunca trabalham simultaneamente, apesar de não existir nenhuma limitação de natureza eletrônica
  - Programa sequencial não consegue ocupar ambos

# Programa Sequencial

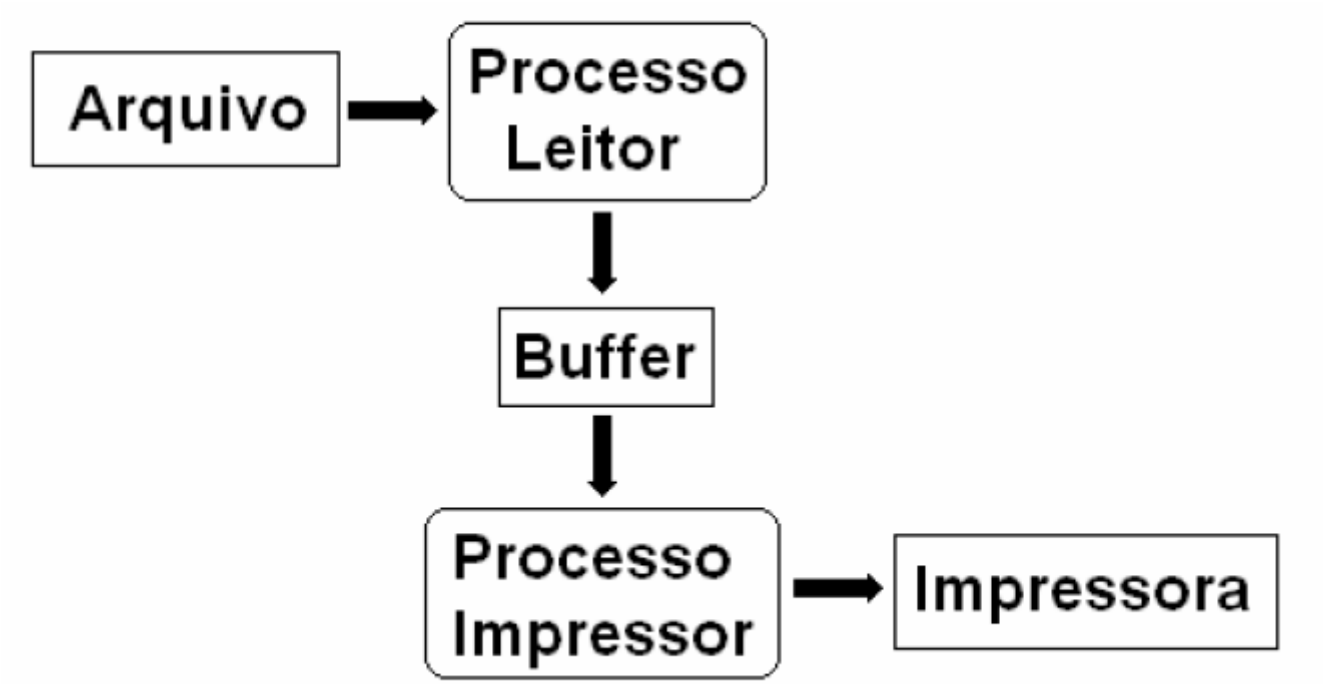




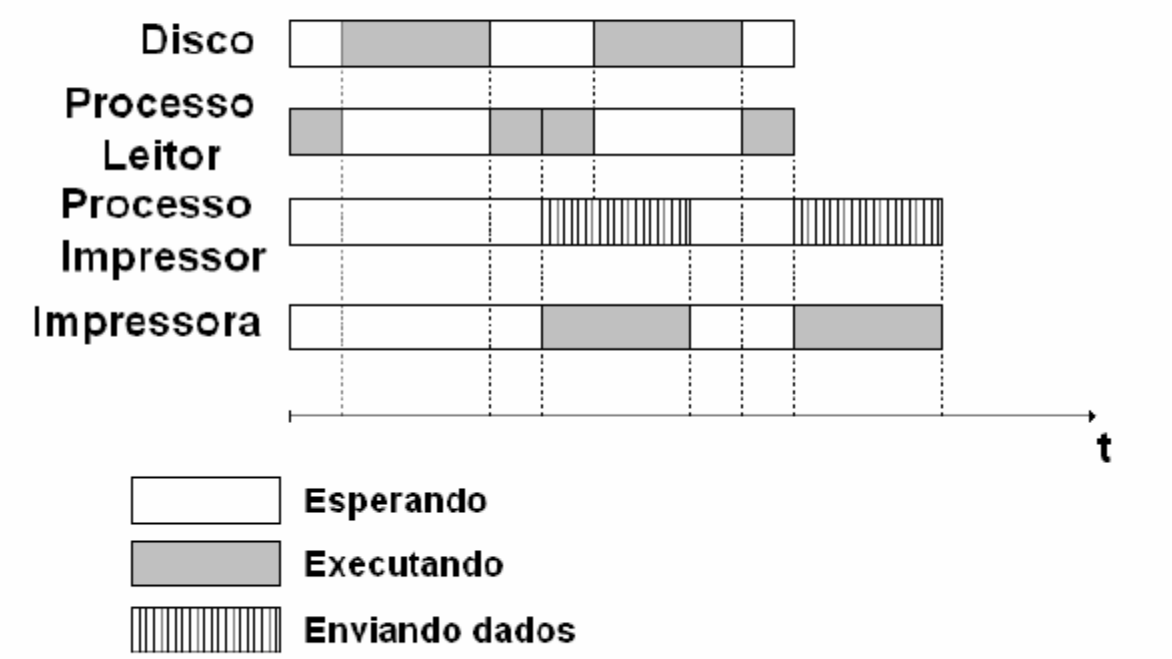
# Programa Concorrente

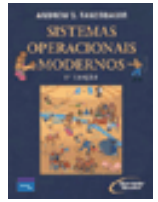
- Dois processos
  - Leitor
    - Lê arquivo, formata e coloca em um *buffer* na memória
  - Impressor
    - Retira dados do *buffer* e envia para a impressora
  - *Obs.: Buffer é uma r*egião de memória temporária utilizada para escrita e leitura de dados (software ou hardware)

# Programa Concorrente



# Programa Concorrente





# Programa Concorrente

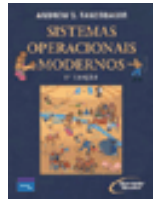
- Disco e impressora trabalham simultaneamente
- Limitações
  - Se processo leitor for sempre mais rápido, o buffer ficará cheio e então o processo leitor terá que esperar até que o processo impressor retire algo do buffer (buffer overflow)
  - Se processo impressor for sempre mais rápido, buffer ficará vazio e terá que esperar pelo processo leitor (*buffer underrun*)

# Especificação do Paralelismo

- Para construir um programa concorrente, antes de mais nada, é necessário ter a capacidade de especificar o paralelismo dentro do programa
- É necessário especificar
  - Quantos processos farão parte do programa
  - Quais rotinas cada um executará

# Parbegin / Parend

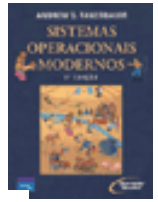
- Delimitam todos os comandos que serão executados em paralelo
- Parbegin (Parallel begin)
  - Comando indicando que a execução sequencial passa a ser dividida em várias sequencias de execução em paralelo
  - Marca o início da execução paralela
- Parend (Parallel end)
  - Comando indicando certas seqüencias de execução paralelas devem se juntar para a execução sequencial continuar
  - Marca o fim da execução paralela



# Parbegin / Parend

- Exemplo

```
Processo A:
    ...
    Parbegin
        comando_1;
        comando_2;
        ...
        comando_n;
    Parend
    ...
    fim
```

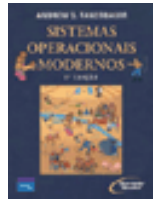


# Especificação do Paralelismo: comentários

- Processos paralelos podem ser executados em qualquer ordem
  - Duas execuções consecutivas do mesmo programa, com os mesmos dados de entrada, podem gerar resultados diferentes
    - Não é necessariamente um erro
  - Possibilidade de forçar a execução em uma determinada ordem

# Compartilhamento de Recursos

- Programação concorrente implica em compartilhamento de recursos
- Processos podem compartilhar todo seu espaço de memória ou apenas uma parte
- Variáveis compartilhadas
  - Processo escreve em uma variável que será lida por outro
  - É necessário controlar o acesso a essas variáveis

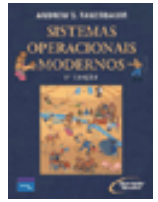


# Seção Crítica

- Parte do código de um processo que realiza a alteração de um recurso compartilhado
- Exemplo:

```
Parbegin
    x=x+1;
    x=x+1;
Parend
```

- Deve-se garantir que nenhum outro processo acesse a seção crítica enquanto um processo o faz



# Seção Crítica

- Uma solução para o problema de seção crítica estará correta quando apresentar as seguintes 4 propriedades:
  - Existe exclusividade mútua entre os processos com referência a execução das respectivas seções críticas
  - Quando um processo P deseja entrar na seção crítica e nenhum outro processo está executando a sua seção crítica, o processo P não é impedido de entrar
  - Nenhum processo pode ter seu ingresso na seção crítica postergado indefinidamente, ou seja, ficar esperando para sempre
  - A solução não depende das velocidades relativas dos processos

# Seção Crítica

- Soluções erradas para o problema da seção crítica normalmente apresentam a possibilidade de postergação indefinida ou a de deadlock
- Postergação indefinida
  - Processo está preso tentando entrar na seção crítica e nunca consegue por ser sempre preterido em benefício de outros processos
- Deadlock
  - Dois ou mais processos estão à espera de um evento que nunca vai ocorrer

# Seção Crítica

```
package semaforos;

public class EstaticoThread extends Thread{

    static float x;
    int idThread;

    public EstaticoThread(int idThread){
        this.idThread = idThread;
    }

    public void run(){
        conta();
    }

    public void conta(){
        if (idThread % 2 == 0){
            for (int i = 1 ; i < 10 ; i++){
                x = (float) (x + Math.pow(idThread, i));
                System.out.println(x);
            }
        } else {
            for (float i = 1 ; i < 10 ; i++){
                x = x + (1 / i);
                System.out.println(x);
            }
        }
    }

}
```

```
package semaforos;

public class Estatico {

    public static void main(String[] args) {
        for (int i = 2 ; i < 4 ; i++){
            Thread t = new EstaticoThread(i);
            t.start();
        }
    }


}
```

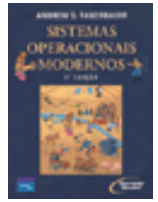
```
1.0
3.5
3.8333333
4.083333
4.283333
4.4499993
4.5928564
4.7178564
4.8289676
3.0
8.828968
16.828968
32.828968
64.828964
128.82896
256.82898
512.829
1024.829
```

# Seção Crítica: soluções

- Desabilitar Interrupções
  - Processo desabilita interrupções antes de acessar variáveis compartilhadas
  - Ao final da seção crítica, processo torna a habilitar as interrupções
  - Utilizado em sistemas pequenos e dedicados a uma única aplicação, sistemas embutidos
    - Ex.: celulares

# Seção Crítica: soluções

- Desabilitar Interrupções
  - Desvantagens
    - Vai contra os mecanismos de proteção do SO
    - Poder demais para processos usuários
    - Diminuição da eficiência do sistema
    - Não funciona em máquinas paralelas



# Seção Crítica: soluções

- Semáforos
  - Tipo abstrato de dado composto por um valor inteiro e uma fila de processos
  - Somente duas operações permitidas no semáforo
    - **P** (testar)
    - **V** (incrementar)

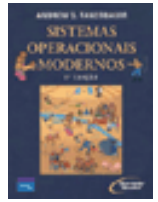
* Todo semáforo deve possuir dois métodos: P e V, que têm sua origem das palavras parsen (passar) e e vrygeren (liberar). Esta definição de semáforo foi proposta por Dijkstra para evitar o tão temido ***DeadLock***

# Seção Crítica: soluções

- Semáforos
  - Quando um processo executa a operação P em um semáforo, o seu valor inteiro é decrementado
  - Caso o novo valor do semáforo seja negativo, o processo é bloqueado e vai para o fim da fila do semáforo

# Seção Crítica: soluções

- Semáforos
  - Quando um processo executa a operação V sobre um semáforo, o seu valor inteiro é incrementado
  - Caso exista algum processo bloqueado na fila desse semáforo, o primeiro processo da fila é liberado



# Seção Crítica: soluções

- Semáforos
  - P(S):

    S.valor = S.valor -1;

    Se S.valor < 0

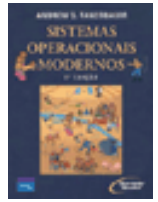
    Então bloqueia o processo , insere em S.fila

  - V(S):

    S.valor = S.valor +1;

    Se S.fila não está vazia

    Então retira processo P de S.fila, acorda P

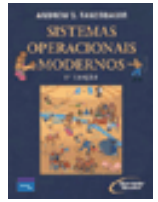


# Seção Crítica: soluções

- Semáforos
  - Para cada estrutura de dados compartilhada, deve ser criado um semáforo S inicializado com o valor 1
  - Todo processo, antes de acessar essa estrutura, deve executar a operação P sobre o semáforo S associado à estrutura de dados em questão
  - Ao sair da seção crítica, o processo executa a operação V sobre o semáforo

# Seção Crítica: soluções

- Semáforos
  - Se valor do semáforo é negativo, significa que existem processos na fila de espera do semáforo
    - Valor absoluto do semáforo é igual ao número de processos na fila de espera



# Seção Crítica: soluções

- Semáforo binário (mutex)
  - Semáforo capaz de assumir apenas os valores 0 e 1
  - Assume apenas os valores livre e ocupado
  - P e V são chamados, respectivamente, de *lock* e *unlock*

*** No Java, ACQUIRE (lock) e RELEASE (unlock)**

# EXEMPLO

```
package semaforo;

import java.util.concurrent.Semaphore;

public class ProcessadorBasico{

      private static Semaphore semaforo;

      private static void processar(int idThread) {
            try {
                System.out.println("Thread #" + idThread + " processando");
                int tempoDormir = (int)(Math.random() * 10000);
                Thread.sleep(tempoDormir);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

      private static void entrarRegiaoNaoCritica(int idThread) {
            System.out.println("Thread #" + idThread + " em região não crítica");
            processar(idThread);
        }

      private static void entrarRegiaoCritica(int idThread) {
            System.out.println("Thread #" + idThread
                    + " entrando em região crítica");
            processar(idThread);
            System.out.println("Thread #" + idThread + " saindo da região crítica");
        }

      public static void processamento(int idThread){
        entrarRegiaoNaoCritica(idThread);
        try {
            semaforo.acquire();
            entrarRegiaoCritica(idThread);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            semaforo.release();
        }
      }
```

```
      static Thread t1 = new Thread(){
          public void run() {
              processamento(1);
          }
      };

      static Thread t2 = new Thread(){
          public void run() {
              processamento(2);
          }
      };

      static Thread t3 = new Thread(){
          public void run() {
              processamento(3);
          }
      };

      static Thread t4 = new Thread(){
          public void run() {
              processamento(4);
          }
      };

      public static void main(String[] args) {
            int numeroDePermicoes = 1;
            semaforo = new Semaphore(numeroDePermicoes);
            t1.start();
            t2.start();
            t3.start();
            t4.start();
        }

}
```

# EXEMPLO

```java
import java.util.concurrent.Semaphore;

public class CarPark {

    public static Semaphore semaforo;

    public static void main(String[] args) {

        int totalCarros = 10;
        int maxCarros = 3;
        semaforo = new Semaphore(maxCarros);
        for (int i = 0 ; i < totalCarros ; i++){
            Thread estacionamento = new CarParkThread(i, semaforo);
            estacionamento.start();
        }
    }

}
```

```java
import java.util.concurrent.Semaphore;

public class CarParkThread extends Thread{

    private int numCarro;
    private Semaphore semaforo;

    public CarParkThread(int numCarro, Semaphore semaforo){
        this.numCarro = numCarro;
        this.semaforo = semaforo;
    }

    public void run(){
        try {
            semaforo.acquire();
            entraCarro();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        finally{
            semaforo.release();
        }
    }

    public void entraCarro(){
        System.out.println("O carro "+numCarro+" entrou");
        ficaCarro();
        saiCarro();
    }

    public void ficaCarro(){
        int tempoEspera = (int)((Math.random() + 1)*5000);
        try {
            Thread.sleep(tempoEspera);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public void saiCarro(){
        System.out.println("O carro "+numCarro+" saiu");
    }

}
```

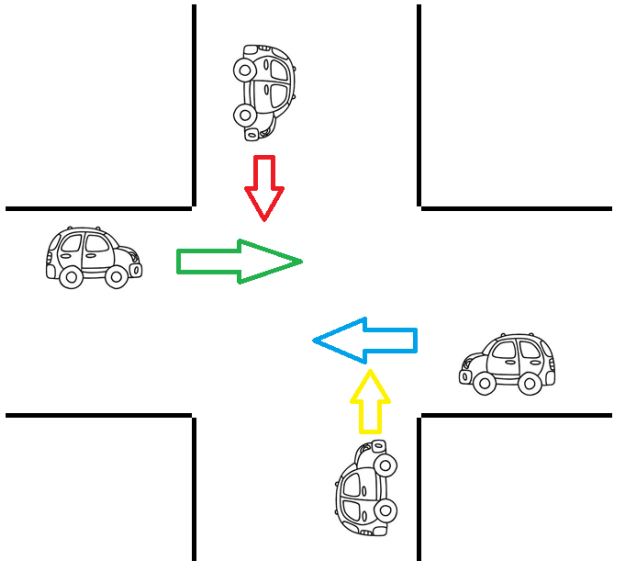
# EXERCÍCIO SALA

4 pessoas caminham, cada uma em um corredor diferente. Os 4 corredores terminam em uma única porta. Apenas 1 pessoa pode cruzar a porta, por vez. Considere que cada corredor tem 200m. e cada pessoa anda de 4 a 6 m/s. Cada pessoa leva de 1 a 2 segundos para abrir e cruzar a porta. Faça uma aplicação em java que simule essa situação.

# EXERCÍCIO

Fazer uma aplicação que gerencie a figura abaixo:

Para tal, usar uma variável sentido, que será alterado pela Thread que controla cada carro com a movimentação do carro. Quando a Thread tiver a possibilidade de ser executada, ela deve imprimir em console o sentido que o carro está passando. Só pode passar um carro por vez no cruzamento.



# EXERCÍCIO

Você foi contratado para automatizar um treino de Fórmula 1. As regras estabelecidas pela direção da provas são simples:

"No máximo 5 carros das 7 escuderias (14 carros no total) presentes podem entrar na pista simultaneamente, mas apenas um carro de cada equipe. O segundo carro deve ficar à espera, caso um companheiro de equipe já esteja na pista. Cada piloto deve dar 3 voltas na pista. O tempo de cada volta deverá ser exibido e a volta mais rápida de cada piloto deve ser armazenada para, ao final, exibir o grid de largada, ordenado do menor tempo para o maior."

# EXERCÍCIO

Um banco deve controlar Saques e Depósitos.

O sistema pode permitir um Saque e um Depósito Simultâneos, mas nunca 2 Saques ou 2 Depósitos Simultâneos.

Para calcular a transação (Saque ou Depósito), o método deve receber o código da conta, o saldo da conta e o valor a ser transacionado.

Deve-se montar um sistema que deve considerar que 20 transações simultâneas serão enviadas ao sistema (aleatoriamente essas transações podem ser qualquer uma das opções) e tratar todas as transações, de acordo com as regras acima.