

Sistemas Operacionais

Cap.6 - Sincronização de Processos



Prof. MSc. Renzo P. Mesquita
renzo@inatel.br

Objetivos

- Introduzir o conceito de Seção Crítica e como o não controle dela pode trazer resultados indesejáveis;
- Apresentar soluções de Software populares para o controle de Seções Críticas;
- Realizar experimentos práticos a fim de melhor compreender a operação de Semáforos e Monitores;



Capítulo 6

Sincronização de Processos

6.1. Introdução

6.2. O Problema da Seção Crítica

6.3. Peterson's Solution

6.4. Semáforos

6.5. Monitores



6.1. Introdução

Como já vimos anteriormente, Processos podem ser de dois tipos: *Independentes ou Cooperantes*.

Processo Cooperante é aquele que pode afetar ou ser afetado por outro Processo.

O que acontece quando vários Processos acessam um recurso compartilhado de forma descontrolada? Dependendo da ordem de acesso, uma variável, por exemplo, pode assumir diversos valores, podendo causar inconsistências no sistema.

A isso damos o nome de *RACE CONDITION*

Vamos ver um exemplo?

6.1. Introdução

EXEMPLO 1

Sejam dois Processos, A e B, que acessam a mesma variável counter de forma concorrente.

O processo A contém a instrução counter++, que nos bastidores pode ser implementada como:

- register1 = counter
- register1 = register1 + 1
- counter = register1

O processo B contém a instrução counter--, que nos bastidores pode ser implementada como:

- register2 = counter
- register2 = register2 - 1
- counter = register2

6.1. Introdução

EXEMPLO 1 (Continuação)

Agora, seja, inicialmente, counter = 5, e a seguinte execução:

- T0: A executa register1 = counter {register1 = 5}
- T1: A executa register1 = register1 + 1 {register1 = 6}
- T2: B executa register2 = counter {register2 = 5}
- T3: B executa register2 = register2 – 1 {register2 = 4}
- T4: A executa counter = register1 {counter = 6}
- T5: B executa counter = register2 {counter = 4}

Como podemos ver, sem um controle, o resultado da execução depende da ordem de acesso à variável counter, podendo causar uma inconsistência.

6.2. O Problema da Seção Crítica

O Problema da Seção Crítica (CriticalSection problem) consiste em seguir um protocolo para que processos possam cooperar entre si quando forem acessar um recurso compartilhado.

A Seção Crítica nada mais é que qualquer recurso computacional (geralmente uma parte de código), que precisa ter o acesso controlado para se evitar a Race Condition .

Durante a execução de um processo, ele pode se encontrar em um dos seguintes momentos:

entry section

Processo disputando uma seção crítica;
critical section

Processo usando uma seção crítica;

exit section

Processo deixando uma seção crítica;
remainder section

Processo executando algo não disputado.

do {

entry section

critical section

exit section

remainder section

} while (TRUE);



6.2. O Problema da Seção Crítica

Independente do protocolo a ser utilizado, para se controlar uma Seção Crítica (Critical Section, ou somente CS), o ideal é que o mesmo obedeça a 3 (três) regras de ouro.

São elas:

- **Exclusão Mútua (Mutual Exclusion):** se um processo P_i está executando em uma CS, então nenhum outro processo pode estar executando nesta CS;
- **Progresso (Progress):** se um processo não está executando na sua CS, então ele não pode impedir que outro processo entre nela;
- **Espera Limitada (Bounded Waiting):** todo processo deve possuir um tempo limite de espera. Se ele precisa usar uma CS, então dentro de um tempo ele tem que usar esta CS;



6.3. Peterson's Solution

Soluções (Protocolos) para o problema da Critical Section podem ser implementadas tanto em Software quanto em Hardware. Nesta disciplina, focaremos nas soluções mais populares, que são as de Software.

O Peterson's Solution é uma solução em Software para o problema da Critical Section, por exemplo, para o caso de haver dois processos cooperantes:

```
//flag[2] é booleana; e turn é um inteiro
flag[0] = 0;
flag[1] = 0;
turn;
```

```
P0: flag[0] = 1;
turn = 1;
while (flag[1] == 1 && turn == 1)
{
    // ocupado, espere
}
// parte crítica
...
// fim da parte crítica
flag[0] = 0;
```

```
P1: flag[1] = 1;
turn = 0;
while (flag[0] == 1 && turn == 0)
{
    // ocupado, espere
}
// parte crítica
...
// fim da parte crítica
flag[1] = 0;
```

- Ao executar o código, observe que, enquanto um processo usa da seção crítica, o outro fica aguardando (bloqueado) dentro do while;
- Apesar de ser simples e efetiva, é uma solução limitada ao controle de uma seção crítica que envolva apenas 2 processos.

6.4. Semáforos

Já uma alternativa popular para controle de Seções Críticas e capaz de controlar o acesso de N processos ao mesmo tempo é fazendo uso de **SEMÁFOROS**

Semáforo é uma ferramenta de sincronização, também de Software, composta por uma variável inteira, acessada somente (fora a inicialização) por duas operações atômicas: wait e signal.

```
1)  wait (S) {  
        while (S <= 0);          // no-op  
        S--;  
    }  
    2)  signal (S) {  
            S++;  
        }
```

6.4. Semáforos

Os Sistemas Operacionais costumam fazer a distinção entre *Semáforos Binários e de Contagem*.

Semáforos Binários

- O valor de um Semáforo Binário só pode variar entre 0 e 1;
- São conhecidos como locks mutex, já que são locks que fornecem exclusão mútua (mutual exclusion);
- N processos compartilham um semáforo (mutex), inicializando com 1;
- Cada Processo Pi é organizado como mostrado na figura abaixo:

```
semaphore mutex = 1; // o semáforo é criado só uma vez!
do {
    wait (mutex);
        // critical section
    signal (mutex);
        // remainder section
} while (TRUE);
```

6.4. Semáforos

EXEMPLO 2

Sejam dois Processos, P1 e P2, e um Semáforo S inicializado em 1. Observe o comportamento do semáforo em cada momento.

| Momento | P1 | P2 | S=1 |
|---------|---|--|-----|
| 1 | Executando um recurso não crítico | | |
| 2 | Executando um recurso não crítico | | |
| 3 | Chama wait() e entra na Seção Crítica (S = 0) | Executando um recurso não crítico | S=0 |
| 4 | Executando na Seção Crítica | | |
| | S=0 | Chama wait(), tenta entrar na Seção Crítica, mas não consegue, pois S = 0 (fica no while) | |
| | | | |
| 4 | Sai da Seção Crítica e chama signal(), ou seja, S = 1 | Agora consegue sair do while e entra na Seção Crítica (S = 0) | P2 |

Neste momento nenhum dos dois processos tentam usar da Seção Crítica;

Neste momento P1 entra na Seção Crítica, decrementa S, mas P2 ainda não tentou entrar na Seção Crítica;

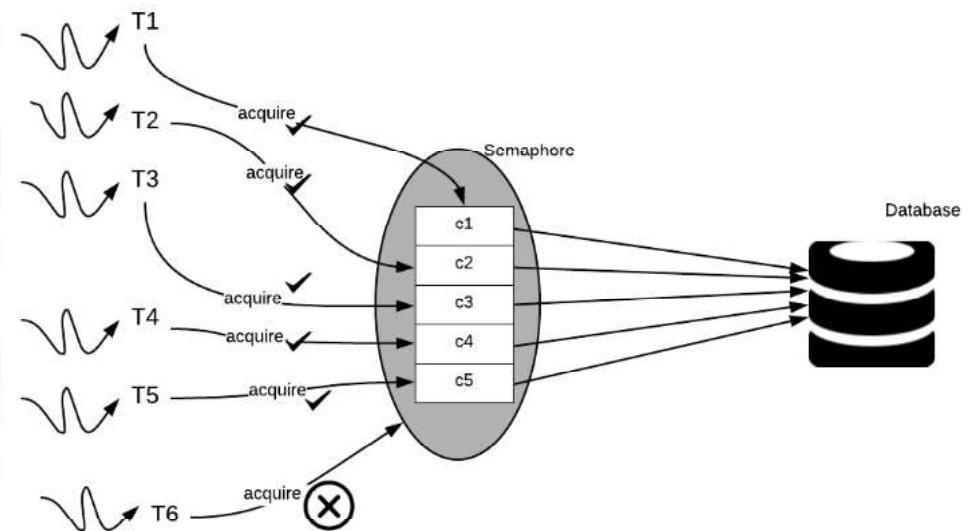
Neste momento P1 está usando da Seção Crítica. P2 tenta entrar, mas não consegue, pois S = 0. Por conta disso, fica travado no while() até S voltar a ser 1;

Neste momento P1 sai da Seção Crítica, e incrementa S. Agora, com a Seção Crítica disponível, P2 consegue entrar para usá-la;

6.4. Semáforos

Semáforos de Contagem

- Funcionamento muito parecido com o Semáforo Binário;
- Porém, são usados para controlar o acesso a um determinado recurso composto por uma quantidade finita de instâncias ($S > 1$);
- O semáforo é iniciado com a quantidade de recursos disponíveis.
- Cada processo que deseja usar um recurso executa uma operação `wait()` no semáforo (decrementando a contagem);
- Quando um processo libera um recurso, ele executa uma operação `signal()`;
- E claro, quando $S = 0$, a Seção Crítica já não consegue mais atender, forçando Processos a esperarem, como é o caso da Thread 6 (T6) no exemplo ao lado. Ele ilustra o acesso a um Banco de Dados sendo gerenciado por um Semáforo de Contagem.



6.4. Semáforos

Apesar da simplicidade e alta aplicabilidade, a principal desvantagem do semáforo é um conceito chamado de Busy Waiting (ou **Espera em Ação**).

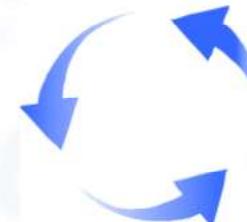
Espera em Ação

Enquanto um Processo estiver fazendo uso de uma Seção Crítica, qualquer outro que tentar entrar na mesma Seção deverá percorrer um looping até o semáforo ser liberado.

- A espera em ação desperdiça ciclos de CPU que algum outro processo poderia usar produtivamente, principalmente em computadores com um só processador;
- Semáforos que usam Espera em Ação são chamados de **SpinLock** porque o processo "gira" enquanto espera o lock;

O que podemos fazer para resolver este problema?

Uma alternativa poderia ser bloquear o processo (suspendê-lo enquanto espera). Porém, ainda assim haverá um overhead necessário para bloquear o processo, e quando o recurso ficar disponível, acordá-lo para fazer uso.

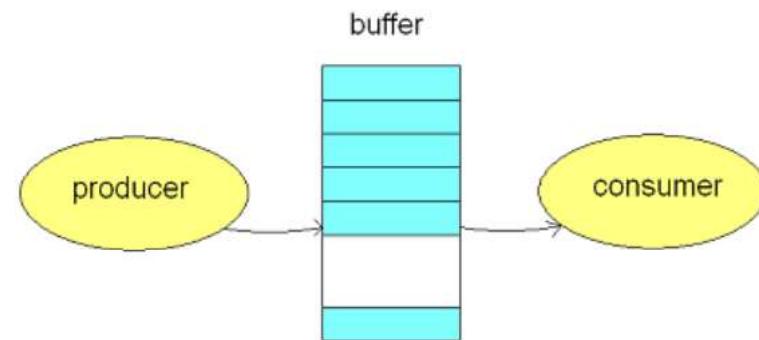


6.4. Semáforos

Problemas Clássicos Resolvidos com Semáforos

1) Problema do Buffer Limitado

- Buffer Limitado, Processo Produtor e Processo Consumidor;
- Um Processo Consumidor só pode Consumir quando o Processo Produtor tiver produzido algo dentro do buffer;



2) Problema dos Leitores/Gravadores

- Controlar a escrita em um recurso (arquivo, variável, etc..) compartilhado;
- Muitas vezes não há necessidade de controle entre Processos que só façam operações de leitura em um recurso;



6.4. Semáforos

3) Problema do Jantar dos Filósofos

O problema do Jantar dos Filósofos é um **problema genérico e abstrato** que é utilizado para explicar diversos problemas que podem acontecer na hora de sincronizar o acesso a recursos pelos Processos.

Ele funciona da seguinte maneira:

- 5 Filósofos, 5 Palitos e uma tigela de arroz;
- Os Filósofos podem executar duas ações: Comer e Pensar;
- Cada filósofo só pode comer quando tem dois palitos (recursos) mais adjacentes disponíveis;

Ou seja, ilustra um exemplo claro de que, muitas vezes, mais de um semáforo (palitos) podem ser necessários para se fazer uso de uma Seção Crítica (neste caso, uma tigela de arroz, mas no mundo da programação, vários outros recursos compartilhados).



```
do {  
    wait ( chopstick[i] );  
    wait ( chopStick[ (i+ 1) % 5 ] );  
    // come  
    signal ( chopstick[i] );  
    signal (chopstick[ (i+ 1) % 5 ] );  
    // pensa  
} while (TRUE);
```

HORA DO Kahoot!



ACESSE:

<https://kahoot.it>

6.4. Semáforos

EXEMPLO 3 (Prática)

Utilizando da linguagem Java, vamos usar do conceito de Semáforos para simular duas situações: um acesso descontrolado e um acesso controlado a uma Impressora (nossa Seção Crítica).

1. Criando uma Classe para representar a Fila de Impressão;

```
public class FilaImpressora {  
    // Método que Imprime Sem Semáforo  
    public void imprimirSemSemaforo() {  
        System.out.println(Thread.currentThread().getName()  
            + " está imprimindo!");  
        System.out.println("Documento da "  
            + Thread.currentThread().getName()  
            + " impresso com sucesso!");  
    }  
  
    public class ProcessoImpressora implements Runnable{  
        // Fila da Impressora  
        FilaImpressora fila;  
  
        // Recebendo a fila que a Thread vai disputar recurso  
        public ProcessoImpressora(FilaImpressora f) {  
            fila = f;  
        }  
  
        @Override  
        public void run() {  
            System.out.println(Thread.currentThread().getName()  
                +" pronta para imprimir!");  
            fila.imprimirSemSemaforo();  
        }  
    }  
}
```

2. Criando uma Classe para representar as Threads que vão disputar a Seção Crítica;

6.4. Semáforos

EXEMPLO 3 (Prática)

3. Na Classe principal, criando os objetos da Fila, das Threads e as colocando para executar;

```
run:  
Thread-1 pronta para imprimir!  
Thread-0 pronta para imprimir!  
Thread-0 está imprimindo!  
Thread-2 pronta para imprimir!  
Documento da Thread-0 impresso com sucesso!  
Thread-1 está imprimindo!  
Thread-2 está imprimindo!  
Documento da Thread-1 impresso com sucesso!  
Documento da Thread-2 impresso com sucesso!  
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

```
public class ExemploSemaforo {  
    public static void main(String[] args) {  
        // CRIANDO A FILA DA IMPRESSORA  
        FilaImpressora fila = new FilaImpressora();  
  
        // CRIANDO UM ARRAY DE 3 THREADS  
        // COLOCANDO TODAS NA MESMA FILA  
        Thread[] thread = new Thread[3];  
        for (int i = 0; i < 3; i++) {  
            thread[i] = new Thread(  
                new ProcessoImpressora(fila));  
        }  
  
        // DANDO START NAS THREADS  
        for (int i = 0; i < 3; i++) {  
            thread[i].start();  
        }  
    }  
}
```

4. Execute a aplicação algumas vezes e observe a desordem na impressão em alguns momentos;

6.4. Semáforos

EXEMPLO 3 (Prática)

Veja que no caso anterior, como não controlamos o acesso ao método imprimir com Semáforos, abrimos brecha para acontecer a Race Condition.

5. Agora adicione o método `imprimirComSemaforo()` dentro da Classe `FilaImpressora` e, dentro da Classe `ProcessoImpressora`, chame ele ao invés de `imprimirSemSemaforo()`;

```
// Criando o Semáforo
// Com S = 1
Semaphore semaforo = new Semaphore(1);

// Método que Imprime Com Semáforo
public void imprimirComSemaforo() {
    try {
        semaforo.acquire(); // wait()
        System.out.println(Thread.currentThread().getName()
            + " está imprimindo!");
    } catch (InterruptedException ex) {}
    finally {
        System.out.println("Documento da "
            + Thread.currentThread().getName()
            + " impresso com sucesso!");
        semaforo.release(); // signal()
    }
}
```

```
Thread-1 pronta para imprimir!
Thread-2 pronta para imprimir!
Thread-0 pronta para imprimir!
Thread-2 está imprimindo!
Documento da Thread-2 impresso com sucesso!
Thread-1 está imprimindo!
Documento da Thread-1 impresso com sucesso!
Thread-0 está imprimindo!
Documento da Thread-0 impresso com sucesso!
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

6. Execute a aplicação algumas vezes e observe a sincronia na hora de executar o código que se encontra dentro do método `imprimirComSemaforo()`;

6.5. Monitores

Semáforos puros, conforme a complexidade de um código cresce, podem gerar soluções confusas e mais susceptíveis a erros. Para evitar tais erros, pesquisadores desenvolveram uma construção em linguagem de alto nível para sincronização de processos e acesso a uma Seção Crítica. Tal construção é chamada MONITOR.

- O MONITOR é uma espécie de Classe, ou seja, um conjunto de variáveis, métodos, e estruturas de dados, capaz de fornecer automaticamente ao programador o controle de uma Seção Crítica;
- Somente um processo pode estar ativo dentro do monitor em um determinado momento;
- Outros processos ficam bloqueados até que possam ganhar acesso ao recurso controlado pelo monitor;
- Depende da Linguagem de Programação (ou seja, nem toda linguagem oferece este recurso);
- Um uso popular do Monitor é utilizando da Linguagem de Programação Java.

```
monitor nomeDoMonitor
{
    // variáveis compartilhadas
    ...

    codigo de inicializacao (...)

    ...

}

// métodos
procedure P1 (...)

...
}

procedure Pn(...)

...
}
```

6.5. Monitores

Monitores em Java

- Todo o mecanismo de controle de um Recurso Crítico é feito automaticamente pelo Compilador do Java;
- Um Monitor Java é uma Classe que possua, ao menos, um método do tipo synchronized;
- O Monitor permite a apenas uma Thread, por vez, executar o método synchronized sobre um objeto desta Classe;
- Todo objeto em Java tem um lock associado a ele. Quando um método synchronized é chamado, o objeto é travado, pois o método recebe o lock do objeto;

Ex: public class SimpleClass

```
  {  
    ...  
    public synchronized void nomeDoMetodo(){  
        // Implementacao do metodo  
    }  
}
```

6.5. Monitores

EXEMPLO 4 (Prática)

Utilizando da linguagem Java, vamos usar do conceito de Monitores para simular duas situações: um acesso descontrolado e um acesso controlado a uma Calculadora (nossa Seção Crítica).

1. Criando uma Classe para representar a Calculadora com pelo menos uma operação de soma (sem controle algum de Seção Crítica);

```
public class Calculadora {  
    // Variavel que vai guardar a soma  
    private int soma;  
  
    // Metodo que faz uma soma  
    public int somaArray(int[] array)  
{  
        soma = 0;  
        for (int i = 0; i < array.length; i++) {  
            soma = soma + array[i];  
  
            System.out.println("Executando a soma " +  
                Thread.currentThread().getName() +  
                " - Somando o valor "+array[i]+  
                " com o total de "+soma);  
  
            try {  
                Thread.sleep(500);  
            } catch (InterruptedException ex) {}  
        }  
        return soma;  
    }  
}
```

6.5. Monitores

EXEMPLO 4 (Prática)

2. Criando uma Classe para representar as Threads que vão disputar a Seção Crítica;

```
public class ThreadSoma implements Runnable{
    // ATRIBUTOS
    public String nome;
    public int[] numeros;
    // CALCULADORA COMPARTILHADA
    // (ps: veja o static)
    private static Calculadora calc = new Calculadora();

    public ThreadSoma(String name, int[] nums) {
        nome = name;
        numeros = nums;
    }

    @Override
    public void run() {

        System.out.println(nome+" iniciada..");
        int soma = calc.somaArray(numeros);
        System.out.println("Resultado da Soma para a "
            +nome+": "+soma);
        System.out.println(nome+" ..finalizada!");
    }
}
```

6.5. Monitores

EXEMPLO 4 (Prática)

3. Na Classe principal, criando os Arrays de entrada, as Threads e as colocando para executar;

```
public class ExemploMonitor {  
    public static void main(String[] args) {  
        // Criando Arrays de inputs  
        int[] sInputs = {10,20,30,40}; //100  
        int[] sInputs2 = {10,20,30,40,50}; //150  
  
        // Criando e iniciando a Thread 1  
        ThreadSoma ts = new ThreadSoma("Thread 1", sInputs);  
        Thread t1 = new Thread(ts,ts.nome);  
        t1.start();  
  
        // Criando e iniciando a Thread 2  
        ThreadSoma ts2 = new ThreadSoma("Thread 2", sInputs2);  
        Thread t2 = new Thread(ts2,ts2.nome);  
        t2.start();  
    }  
}
```

```
Thread 2 iniciada..  
Thread 1 iniciada..  
Executando a soma Thread 2 - Somando o valor 10 com o total de 10  
Executando a soma Thread 1 - Somando o valor 10 com o total de 10  
Executando a soma Thread 1 - Somando o valor 20 com o total de 50  
Executando a soma Thread 2 - Somando o valor 20 com o total de 50  
Executando a soma Thread 1 - Somando o valor 30 com o total de 80  
Executando a soma Thread 2 - Somando o valor 30 com o total de 80  
Executando a soma Thread 1 - Somando o valor 40 com o total de 160  
Executando a soma Thread 2 - Somando o valor 40 com o total de 160  
Executando a soma Thread 2 - Somando o valor 50 com o total de 210  
Resultado da Soma para a Thread 1: 160  
Thread 1 ..finalizada!  
Resultado da Soma para a Thread 2: 210  
Thread 2 ..finalizada!  
CONSTRUÍDO COM SUCESSO (tempo total: 2 segundos)
```

4. Execute a aplicação algumas vezes e observe a desordem na impressão em alguns momentos;



6.5. Monitores

EXEMPLO 4 (Prática)

Veja que no caso anterior, como o objeto compartilhado da Calculadora não se comportou como um Monitor, abrimos brecha para acontecer a Race Condition.

5. Agora simplesmente adicione a palavra chave synchronized no cabeçalho do método de soma para transformar a Calculadora em um Monitor;

```
Thread 1 iniciada..
Thread 2 iniciada..
Executando a soma Thread 1 - Somando o valor 10 com o total de 10
Executando a soma Thread 1 - Somando o valor 20 com o total de 30
Executando a soma Thread 1 - Somando o valor 30 com o total de 60
Executando a soma Thread 1 - Somando o valor 40 com o total de 100
Executando a soma Thread 2 - Somando o valor 10 com o total de 10
Resultado da Soma para a Thread 1: 100 ←
Thread 1 ..finalizada!
Executando a soma Thread 2 - Somando o valor 20 com o total de 30
Executando a soma Thread 2 - Somando o valor 30 com o total de 60
Executando a soma Thread 2 - Somando o valor 40 com o total de 100
Executando a soma Thread 2 - Somando o valor 50 com o total de 150
Resultado da Soma para a Thread 2: 150 ←
Thread 2 ..finalizada!
CONSTRUÍDO COM SUCESSO (tempo total: 4 segundos)
```

```
// Método que faz uma soma
public synchronized int somaArray(int[] array)
{
    soma = 0;
    for (int i = 0; i < array.length; i++) {
        soma = soma + array[i];
    }
    ...
}
```

6. Execute a aplicação algumas vezes e observe a sincronia na hora de executar a soma em cada uma das Threads;

6.5. Monitores

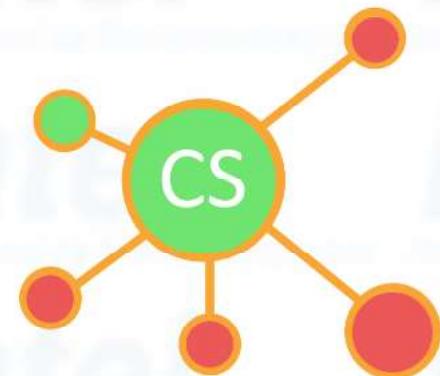
TRABALHO PRÁTICO 3

Agora que você já conhece os conceitos fundamentais de Semáforos e Monitores, crie um exemplo simples de aplicação em Java (de preferência) ou na linguagem da sua escolha, que seja capaz de demonstrar:

- *O uso de uma Seção Crítica sem controle;*
- *O uso de uma Seção Crítica com controle;*

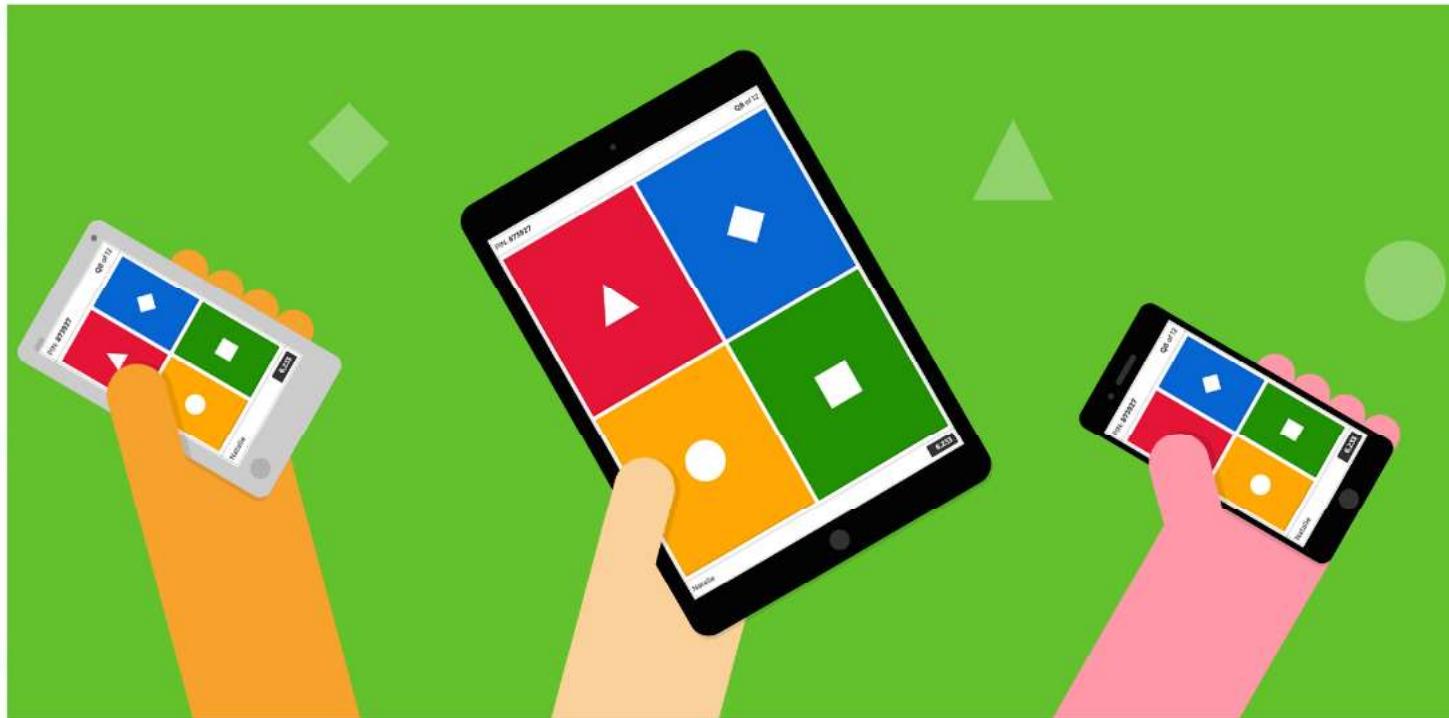
IMPORTANTE:

- O trabalho por ser feito usando de Semáforos OU Monitores;
- Use de seus conhecimentos e criatividade :)
- O Trabalho pode ser individual ou em dupla;
- Fique atento à data de apresentação proposta pelo professor.



HORA DO

Kahoot!



ACESSE:

<https://kahoot.it>

**FIM
DO
CAPÍTULO 6**



EXERCÍCIOS