**FACULDADE DE COMPUTAÇÃO E INFORMÁTICA**
**BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO**
**SISTEMAS OPERACIONAIS – Aula 06 – 2º SEMESTRE/2019**
**PROF. LUCIANO SILVA**

## TEORIA: SEMÁFOROS E ALOCAÇÃO DE RECURSOS A PROCESSOS

Nossos objetivos nesta aula são:

- conhecer o conceito de semáforo e sua importância para controlar acesso concorrente a recursos
- conhecer e estudar técnicas de detecção de *deadlock*
- conhecer e estudar técnicas de prevenção de *deadlock*

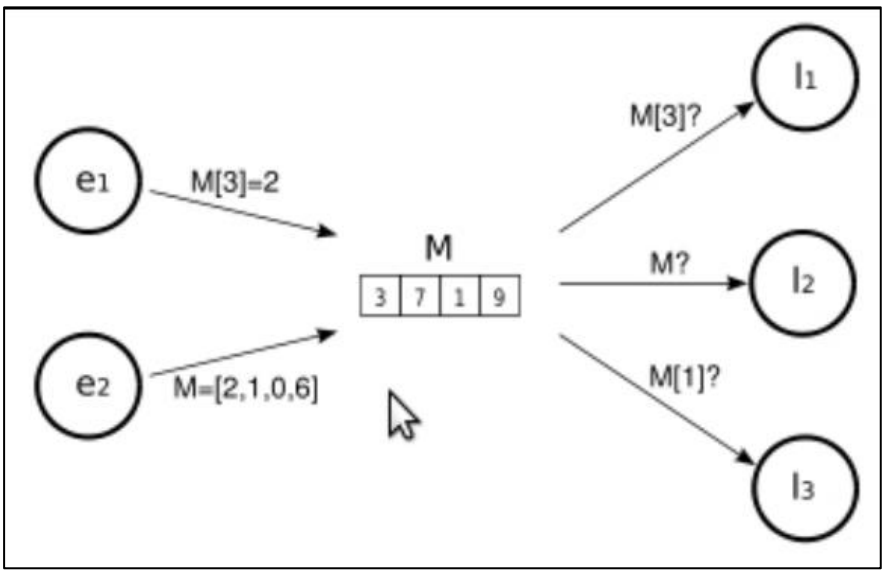

Para esta aula, usamos como referência as **Seções 5.7.4 e 5.8** do nosso livro-texto:

STUART, B.L**. Princípios de Sistemas Operacionais: Projetos e Aplicações**. São Paulo: Cengage Learning, 2011.

*Não deixem de ler estas seções depois desta aula!*

## SEMÁFOROS

- Em diversas situações anteriores, tivemos a oportunidade de identificar **acesso concorrente** de processos e threads a determinados recursos compartilhados. Um exemplo bastante recente foi no problema do produtor-consumidor, onde o consumidor poderia tentar o acesso na mesma posição em que o produtor estivesse escrevendo.



- No exemplo anterior, dependendo **quem tenha acesso inicial** ao buffer M (processos e1 ou e2), teremos **valores diferentes** de M lidos pelos processos (I1, I2 e I3).

- Vamos adotar uma política bem rígida, chamada **Política da Exclusão Mútua**: *se um processo tiver ganho o uso de um recurso, outro processo não poderá utilizar este recurso até que o recurso seja liberado.*

- Para implementar a Política da Exclusão Mútua, podemos utilizar estruturas de dados chamadas **semáforos**. Um semáforo possui uma **variável de controle** que pode ser **incrementada** ou **decrementada**, além de uma fila de espera por este semáforo. Disponibilizaremos duas funções: **up() e down().**

- A implementação via semáforo segue a seguinte política:

    - Inicializamos o semáforo com **um**.

    - **up()**: somamos um no valor da variável de controle do semáforo. Removemos o primeiro processo da fila do semáforo e damos o controle do recurso a ele.

    - **down()**: se a variável de controle for igual a zero, colocamos o processo na fila do semáforo. Subtraímos 1 na variável de controle do semáforo.

## EXERCÍCIO COM DISCUSSÃO EM DUPLAS

No exemplo abaixo da aula passada, temos os processos produtor e consumidor. Ambos os processos possuem problemas de acesso concorrente a à variável compartilhada **shared_buff**.

```
item nextProduced;

while(1){

  // Se não há espaço, fica esperando…..
  while((free_index+1) % buff_max == full_index);

  shared_buff[free_index] = nextProduced;
  free_index = (free_index + 1) % buff_max;
}
```

**PROCESSO-PRODUTOR**

```
item nextConsumed;

while(1){

  // Se não há item produzido, espera…
  while((free_index == full_index);

  nextConsumed = shared_buff[full_index];
  full_index = (full_index + 1)%buff_max;
}
```

**PROCESSO-CONSUMIDOR**

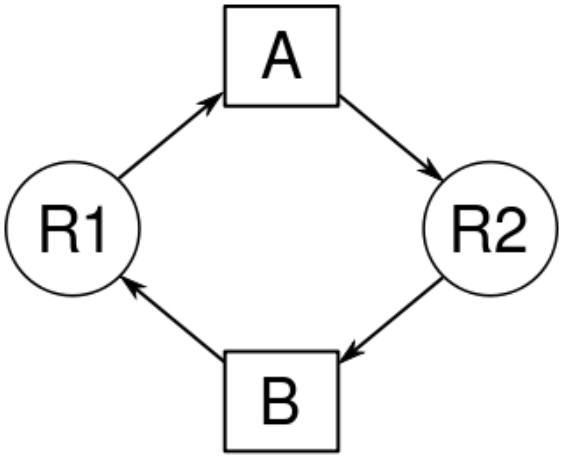
Resolva o problema de acesso concorrente à variável **shared_buff** usando **semáforos**.

## DEADLOCK

- Dizemos que um conjunto de processos está em situação de ***deadlock*** quanto não conseguem mais prosseguir suas execuções por dependerem de recursos alocados a outros processos e que não podem mais ser liberados.

- Vamos considerar a seguinte situação (nesta ordem):

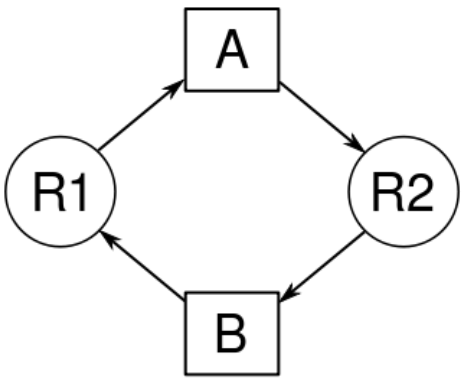
    - O processo A ganhou o uso do recurso R1.
    - O processo B ganhou o uso do recurso R2.
    - O processo A solicita uso do recurso R2, que está alocado a B. **Logo A será bloqueado até que B libere R2.**
    - O processo B solicita uso do recurso R1, que está alocado a A. **Logo B será bloqueado até que A libere R1.**

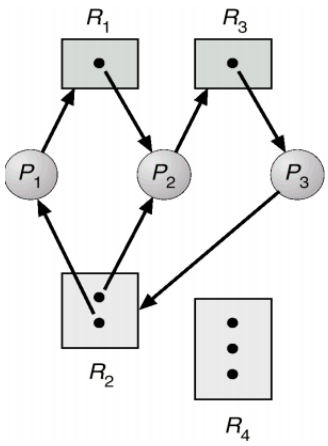



- Neste caso, estamos em ***deadlock***. O processo A só consegue continuar a execução se B liberarR2. B só consegue continuar a execução de A liberar R1. Logo, nem A e nem B conseguem continuar a execução.

## EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Observando a figura à esquerda, qual seria uma **condição necessária e suficiente** para que um grafo que represente esta situação esteja em *deadlock* ?



## ALGORITMO PARA DETECÇÃO DE DEADLOCK

- Para detectar um possível estado de deadlock, vamos supor que tenhamos n processos e m recursos. Vamos utilizar um vetor (**Available**) e duas matrizes(**Allocation** e **Request**), que irão compor o estado do sistema num determinado instante:
  - **Available**: vetor de tamanho m com os recursos disponíveis.
  - **Allocation**: matriz de tamanho nxm com os recursos alocados para os processos
  - **Request**: matriz de tamanho nxm com os recursos solicitados pelos processos.

1. **Work** e **Finish** são vetores de tamanho M e N respectivamente
   - Work= Available
   - Finish[i]= false, se $Allocation_i$!= 0, senão Finish[i]= true
2. Procurar um i, onde
   - Finish[i]== false
   - $Request_i$<= Work
   - Se um nenhum elemento i existe com essas condições, vá para o passo 4
3. Passo 3
   - Work= Work + Allocation
   - Finish[i]= true
   - Vá para o passo 2
4. Passo4
   - Se existe Finish[i] ==false, então o sistema está em estado de impasse. Os processos com Finish=false, estão em impasse.

## EXERCÍCIO TUTORIADO

Verificar, usando o algoritmo acima, se o estado abaixo configura um estado de deadlock. Caso afirmativo, mostrar quais processos estão em deadlock.

**Recursos**: A (7 instâncias), B (2 instâncias) e C (6 instâncias)

| | *Allocation* | *Request* |
|---|---|---|
| | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 0 0 0 |
| $P_1$ | 2 0 0 | 2 0 2 |
| $P_2$ | 3 0 3 | 0 0 0 |
| $P_3$ | 2 1 1 | 1 0 0 |
| $P_4$ | 0 0 2 | 0 0 2 |

| *Available* |
|---|
| *A B C* |
| 0 0 0 |

**EXERCÍCIO COM DISCUSSÃO EM DUPLAS**

Verificar, usando o algoritmo acima, se o estado abaixo configura um estado de deadlock. Caso afirmativo, mostrar quais processos estão em deadlock.

**Recursos**: A (7 instâncias), B (2 instâncias) e C (6 instâncias)

| | *Allocation* | *Request* |
|---|---|---|
| | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 0 0 0 |
| $P_1$ | 2 0 0 | 2 0 2 |
| $P_2$ | 3 0 3 | 0 0 1 |
| $P_3$ | 2 1 1 | 1 0 0 |
| $P_4$ | 0 0 2 | 0 0 2 |

| *Available* |
|---|
| *A B C* |
| 0 0 0 |

## PREVENÇÃO DE DEADLOCK

- Prever a ocorrência de deadlock é muito melhor do que deixar o S.O. chegar a uma situação de deadlock. Veremos, em seguida, o **Algoritmo do Banqueir**o (Banker´s Algorithm), cuja idéia central é só alocar recursos a um processo desde que estejamos num estado seguro.

- Supondo novamente que tenhamos n processos e m recursos, faremos uso de um vetor (Available) e três matrizes (Max, Allocation e Need):
  - **Available**: vetor de tamanho m com os recursos disponíveis.
  - **Max**: matriz de tamanho nxm com número máximo de recursos que cada processo por requisitar.
  - **Allocation**: matriz de tamanho nxm com os recursos já alocados pelos processos.
  - **Need**: matriz de tamanho nxm com os recursos que o processo ainda necessita para terminar.

**PASSO 1:** VERIFICAR SE SISTEMA ESTÁ EM ESTADO SEGURO

1. **Work** e **Finish** são dois vetores de tamanho m e n, respectivamente. Inicializar
   - **Work= available**
   - **Finish[i]= false, para i=0..N**
2. Procure um i, onde:
   - **Finish[i]= false**
   - **$Need_i$<=Work**
   - **Se não existir um processo i, vá para o passo 4**
3. Passo 3
   - **Work= Work+ $Allocation_i$**
   - **Finish[i]= true**
   - **Volte para o passo 2**
4. Passo 4
   - **Se Finish[i]=true para todo i, o sistema está em estado seguro**

**PASSO 2:** ALOCAR OS RECURSOS

- **$Request_i$**= vetor de requisições do processo Pi

1. **Se $Request_i$ <= $Need_i$**, vá para o passo 2. Senão, sinalize um erro, pois o processo solicitou mais recursos que ele declarou necessitar
2. **Se $Request_i$ <= Available,** vá para o passo 3. Senão, o processo Pi deve esperar pois os recursos não estão disponíveis
3. O processo Pi pretende alocar os recursos, logo é necessário atualizar as estruturas de dados

$$Available = Available - Request;$$
$$Allocation_i = Allocation_i + Request_i;$$
$$Need_i = Need_i - Request_i;$$

   - **Se seguro => os recursos podem ser alocados**
   - **Senão o processo deve esperar, e o estado anterior deve ser restaurado**

## EXERCÍCIO TUTORIADO

Verifique se o processo P1 conseguiria alocar o vetor de requisição **Request=(1,0,2)**, utilizando o Algoritmo do Banqueiro.

**Recursos**: A (10 instâncias), B (5 instâncias) e C (7 instâncias)

| | *Allocation* | *Max* |
|---|---|---|
| | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 |
| $P_1$ | 2 0 0 | 3 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 |
| $P_3$ | 2 1 1 | 2 2 2 |
| $P_4$ | 0 0 2 | 4 3 3 |

*Available*

*A B C*

3 3 2

## EXERCÍCIO COM DISCUSSÃO EM DUPLAS

Verifique se o processo P4 conseguiria alocar o vetor de requisição **Request=(3,3,0)**, utilizando o Algoritmo do Banqueiro.

**Recursos**: A (10 instâncias), B (5 instâncias) e C (7 instâncias)

| | *Allocation* | *Max* |
|---|---|---|
| | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 |
| $P_1$ | 2 0 0 | 3 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 |
| $P_3$ | 2 1 1 | 2 2 2 |
| $P_4$ | 0 0 2 | 4 3 3 |

*Available*

*A B C*

3 3 2

## EXERCÍCIOS EXTRA-CLASSE

1. Construa um programa C para representar um semáforo. Reescreva o código do produtor e consumidor usando este código.
2. Verificar se o estado abaixo configura um estado de deadlock. Caso afirmativo, mostrar quais processos estão em deadlock.

   **Recursos**: A (7 instâncias), B (2 instâncias) e C (6 instâncias)

| | *Allocation* | *Request* |
|---|---|---|
| | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 0 0 0 |
| $P_1$ | 2 0 0 | 2 0 2 |
| $P_2$ | 3 0 3 | 0 0 0 |
| $P_3$ | 2 1 1 | 1 0 0 |
| $P_4$ | 0 0 2 | 0 0 2 |

| *Available* |
|---|
| *A B C* |
| 0 0 0 |

3. Verifique se o processo P2 conseguiria alocar o vetor de requisição **Request=(2,0,2)**, utilizando o Algoritmo do Banqueiro.

   **Recursos**: A (10 instâncias), B (5 instâncias) e C (7 instâncias)

| | *Allocation* | *Max* |
|---|---|---|
| | *A B C* | *A B C* |
| $P_0$ | 0 1 0 | 7 5 3 |
| $P_1$ | 2 0 0 | 3 2 2 |
| $P_2$ | 3 0 2 | 9 0 2 |
| $P_3$ | 2 1 1 | 2 2 2 |
| $P_4$ | 0 0 2 | 4 3 3 |

| *Available* |
|---|
| *A B C* |
| 3 3 2 |