URI | UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES

30-769

Linguagem de Programação IV

MSc. Fernando Schubert

SINCRONIZAÇÃO

- É comum um processo ter que esperar até que uma condição se torne verdadeira. Para essa espera ser "eficiente", o processo deve esperar no estado bloqueado (sem competir pela UCP). Dois tipos de bloqueio são considerados básicos:

    1. bloquear até que um recurso se torne disponível
    2. bloquear até que chegue um sinal de outro processo.

SINCRONIZAÇÃO

- Estes bloqueios são caracterizados pelas operações básicas lock/unlock e block/wakeup(P)Enquanto o primeiro par (lock/unlock) implementa sincronização do tipo exclusão mútua, o segundo par implementa uma forma básica de comunicação.
- Um processo só entra num trecho delimitado pelo par lock/unlock se nenhum outro processo está executando em um outro trecho delimitado dessa maneira.
- Isto é, o primeiro processo que executa o comando lock passa e tranca a passagem (chaveia a fechadura) para os demais.
- O comando unlock deixa passar (desbloqueia) o primeiro processo da fila de processos que estão bloqueados por terem executado um lock (enquanto a fechadura estava trancada).
- Se a fila está vazia, a fechadura é destrancada (isto é, é deixada aberta)

SINCRONIZAÇÃO

- Estes bloqueios são caracterizados pelas operações básicas lock/unlock e block/wakeup(P)Enquanto o primeiro par (lock/unlock) implementa sincronização do tipo exclusão mútua, o segundo par implementa uma forma básica de comunicação.

SINCRONIZAÇÃO - *lock / unlock*

- Um processo só entra num trecho delimitado pelo par lock/unlock se nenhum outro processo está executando em um outro trecho delimitado dessa maneira.
- Isto é, o primeiro processo que executa o comando lock passa e tranca a passagem (chaveia a fechadura) para os demais.
- O comando unlock deixa passar (desbloqueia) o primeiro processo da fila de processos que estão bloqueados por terem executado um lock (enquanto a fechadura estava trancada).
- Se a fila está vazia, a fechadura é destrancada (isto é, é deixada aberta)

SINCRONIZAÇÃO

```
public class Account // This is a monitor of an account
{
    private decimal _balance = 0;
    private object _balanceLock = new object();

    public void Deposit(decimal amount)
    {
        // Only one thread at a time may execute this statement.
        lock (_balanceLock)
        {
            _balance += amount;
        }
    }

    public void Withdraw(decimal amount)
    {
        // Only one thread at a time may execute this statement.
        lock (_balanceLock)
        {
            _balance -= amount;
        }
    }
}
```

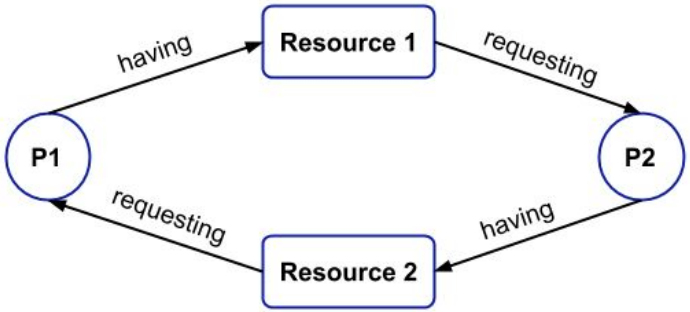
https://learn.microsoft.com/pt-br/dotnet/csharp/language-reference/statements/lock

SINCRONIZAÇÃO - *block/wakeup(P)*

- Quando um processo P executa o comando block, ele se bloqueia até que um outro processo execute o comando wakeup(P).
- Este último comando acorda (desbloqueia) o processo especificado por P.
- Se wakeup(P) é executado antes, o processo P não se bloqueia ao executar o block.
- Na verdade, os comandos lock e unlock não formam uma estrutura sintática (isto é, não precisam estar casados, como se fossem um abre e fecha parênteses), eles são comandos independentes.

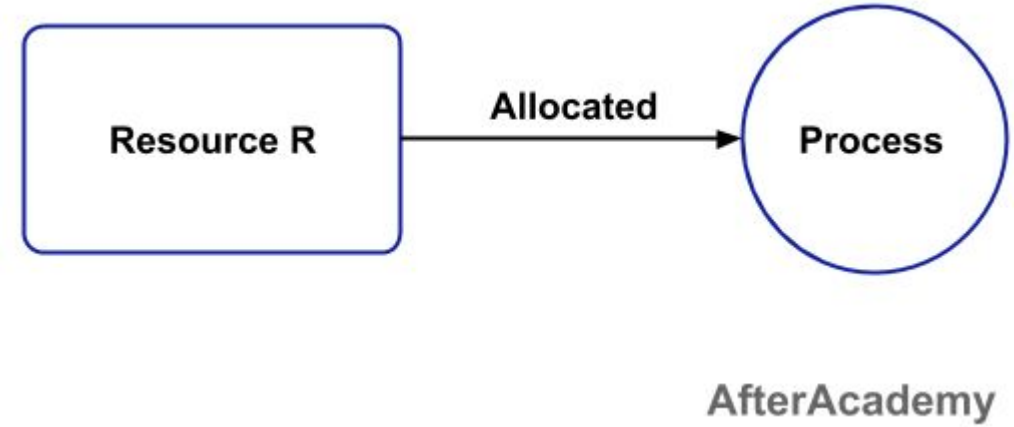
DEADLOCKS

- Deadlock é uma situação em que dois ou mais processos ficam esperando um pelo outro.
- Imagine que temos dois processos, P1 e P2.
  - O processo P1 está segurando o recurso R1 e esperando pelo recurso R2.
  - Ao mesmo tempo, o processo P2 está com o recurso R2 e esperando pelo recurso R1.
  - Nenhum dos processos libera seus recursos, resultando em uma espera infinita e nenhuma tarefa sendo realizada. Isso é chamado de Deadlock.



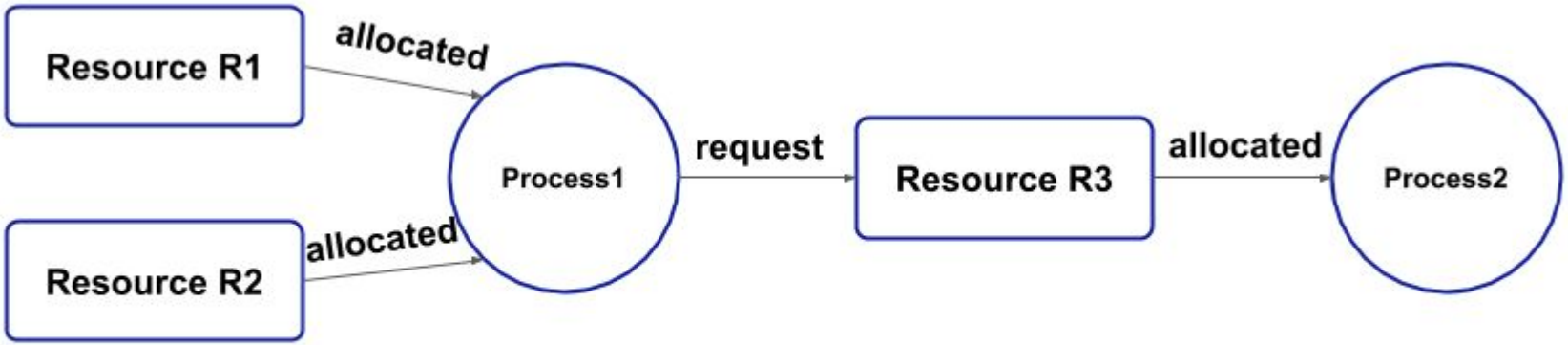
DEADLOCKS - CONDIÇÕES NECESSÁRIAS

- **Exclusão Mútua:** Um recurso só pode ser mantido por um processo de cada vez. Em outras palavras, se um processo P1 está usando um recurso R em um determinado momento, outro processo P2 não pode manter ou usar o mesmo recurso R nesse mesmo instante. O processo P2 pode fazer uma solicitação para usar o recurso R, mas não pode usá-lo simultaneamente com o processo P1.



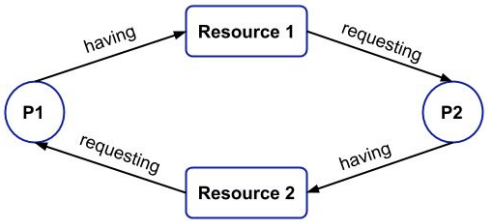
DEADLOCKS - CONDIÇÕES NECESSÁRIAS

- **Espera ocupada:** Um processo pode manter vários recursos ao mesmo tempo e, ao mesmo tempo, solicitar outros recursos que estão sendo mantidos por outro processo. Por exemplo, um processo P1 pode manter dois recursos, R1 e R2, e ao mesmo tempo, solicitar o recurso R3 que está sendo mantido pelo processo P2.



DEADLOCKS - CONDIÇÕES NECESSÁRIAS

- **Espera circular:** A espera circular é uma condição em que o primeiro processo está esperando pelo recurso mantido pelo segundo processo, o segundo processo está esperando pelo recurso mantido pelo terceiro processo, e assim por diante. No final, o último processo está esperando pelo recurso mantido pelo primeiro processo. Assim, cada processo está esperando que o outro libere o recurso, e ninguém libera o seu próprio recurso. Todos estão esperando para obter o recurso. Isso é chamado de espera circular.



SEMÁFOROS

SEMÁFOROS

- Semáforo Binário (ou Mutex): (igual a um *lock*)
  - Garante acesso mutuamente exclusivo a um recurso (apenas um processo pode estar na seção crítica por vez).
  - Pode variar de 0 a 1.
  - É inicializado como livre (valor = 1).

SEMÁFOROS

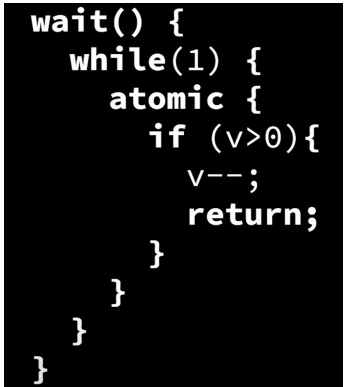
- Semáforo de Contagem:
  - Útil quando múltiplas unidades de um recurso estão disponíveis.
  - A contagem inicial à qual o semáforo é inicializado geralmente corresponde ao número de recursos.
  - Um processo pode adquirir acesso desde que pelo menos uma unidade do recurso esteja disponível.

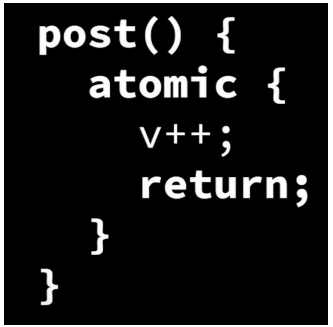
SEMÁFOROS

- Assim como locks, um semáforo suporta duas operações atômicas: `Semaphore->Wait()` e `Semaphore->Signal()`.
  - `S->Wait()` // aguarda até que o semáforo S esteja disponível

    <seção crítica>

  - `S->Signal()` // sinaliza para outros processos que o semáforo S está livre
  - Cada semáforo mantém uma fila de processos que estão esperando para acessar a seção crítica (por exemplo, para comprar leite).
  - Se um processo executa `S->Wait()` e o semáforo S está livre (não zero), ele continua executando. Se o semáforo S não está livre, o sistema operacional coloca o processo na fila de espera para o semáforo S.
  - Um `S->Signal()` desbloqueia um processo na fila de espera do semáforo S.





SEMÁFOROS

- Assim como locks, um semáforo suporta duas operações atômicas: `Semaphore->Wait()` e `Semaphore->Signal()`.
  - `S->Wait()` // aguarda até que o semáforo S esteja disponível

    <seção crítica>

  - `S->Signal()` // sinaliza para outros processos que o semáforo S está livre
  - Cada semáforo mantém uma fila de processos que estão esperando para acessar a seção crítica (por exemplo, para comprar leite).
  - Se um processo executa `S->Wait()` e o semáforo S está livre (não zero), ele continua executando. Se o semáforo S não está livre, o sistema operacional coloca o processo na fila de espera para o semáforo S.
  - Um `S->Signal()` desbloqueia um processo na fila de espera do semáforo S.