FACULDADE
SANTA TEREZINHA

SISTEMAS OPERACIONAIS

PROF CARLOS WAGNER
FACULDADE CEST

UNIDADE II – GERENCIAMENTO DE PROCESSOS (16h)

2.1 Modelos de processos

2.2 Escalonamento

2.3 Sincronização

2.4 Impasses

2.5 Gerenciamento de processos

2.3 - Impasses

- **Deadlocks**

Em um ambiente de multiprogramação, vários processos podem competir por um número finito de recursos. Um processo solicita recursos; se os recursos não estão disponíveis naquele momento, o processo entra em estado de espera. Em alguns casos, um processo em espera não consegue mais mudar de estado novamente porque os recursos que ele solicitou estão reservados para outros processos em espera. Essa situação é chamada deadlock.

2.3 - Impasses

- **Modelos de Sistema**

Um sistema é composto por um número finito de recursos a serem distribuídos entre vários processos competidores.

Exemplos de recursos:

- Ciclos de CPU;
- Arquivos;
- dispositivos de I/O (como impressoras e drives de DVD)

Um processo deve solicitar um recurso antes de usálo e deve liberar o recurso após usálo. O processo pode solicitar tantos recursos quantos precisar para executar sua tarefa. O número de recursos solicitados não pode exceder o número total de recursos disponíveis no sistema.

2.3 - Impasses

- **Modelos de Sistema**

Para usar um recurso, um processo precisa obedecer à seguinte sequência:

1) **Solicitação**. O processo solicita o recurso. Se a solicitação não puder ser atendida imediatamente (por exemplo, se o recurso estiver sendo usado por outro processo), o processo solicitante deve esperar até que possa adquirir o recurso.
2) **Uso**. O processo pode operar sobre o recurso (por exemplo, se o recurso for uma impressora, o processo pode imprimir na impressora).
3) **Liberação**. O processo libera o recurso.

2.3 - Impasses

- **Modelos de Sistema**

A solicitação e liberação dos recursos podem ser feitas através de chamadas de sistemas:

`request()` e **`release()`** - dispositivos

`open()` e **`close()`** - arquivos

`allocate()` e **`free()`** - memória

… ou operações sobre semáforos;

`wait()` e **`signal()`**

.. ou **`acquire()`** e **`release()`** para liberação de mutexes.

2.3 - Impasses

- **Deadlocks com mutex**

A função pthread_mutex_init( ) inicializa um mutex destrancado. Os locks mutex são adquiridos e liberados com o uso de pthread_mutex_lock( ) e pthread_mutex-unlock( ), respectivamente. Se um thread tentar adquirir um mutex trancado (submetido a um lock), uma chamada a pthread_mutex_lock( ) bloqueará o thread até que o proprietário do lock mutex invoque pthread_mutex_unlock( ).

```
/* Cria e inicializa os locks mutex */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;
pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

```
/* thread_one é executado nessa função */
void *do_work_one(void *param)
{
  pthread_mutex_lock(&first_mutex);
  pthread_mutex_lock(&second_mutex);
  /**
   * Executa algum trabalho
   */
  pthread_mutex_unlock(&second_mutex);
  pthread_mutex_unlock(&first_mutex);
  pthread_exit(0);
}
/* thread_two é executado nessa função */
void *do_work_two(void *param)
{
  pthread_mutex_lock(&second_mutex);
  pthread_mutex_lock(&first_mutex);
  /**
    * Executa algum trabalho
    */
  pthread_mutex_unlock(&first_mutex);
  pthread_mutex_unlock(&second_mutex);
  pthread_exit(0);
}
```

2.3 - Impasses

- **Caracterização de um deadlock**

Em um deadlock, os processos nunca terminam sua execução, e os recursos do sistema ficam ocupados, impedindo que outros jobs comecem a ser executados e pode acontecer se quatro condições ocorrerem simultaneamente em um sistema:

- **Exclusão mútua.** Apenas um processo de cada vez pode usar o recurso. Se outro processo solicitar esse recurso, o processo solicitante deve ser atrasado até que o recurso tenha sido liberado.
- **Retenção e espera**. Um processo deve estar de posse de pelo menos um recurso e esperando recursos retidos por outros processos.
- **Inexistência de preempção.** Os recursos só podem ser liberados voluntariamente pelo processo que o estiver retendo, após esse processo ter completado sua tarefa.
- **Espera circular.** Deve haver um conjunto {P0, P1, …, Pn} de processos em espera tal que P0 esteja esperando por um recurso retido por P1, P1 esteja esperando por um recurso retido por P2, …, Pn1 esteja esperando por um recurso retido por Pn, e Pn esteja esperando por um recurso retido por P0.

2.3 - Impasses

- **Grafo de Alocação de Recursos**

Os conjuntos P, R e E:

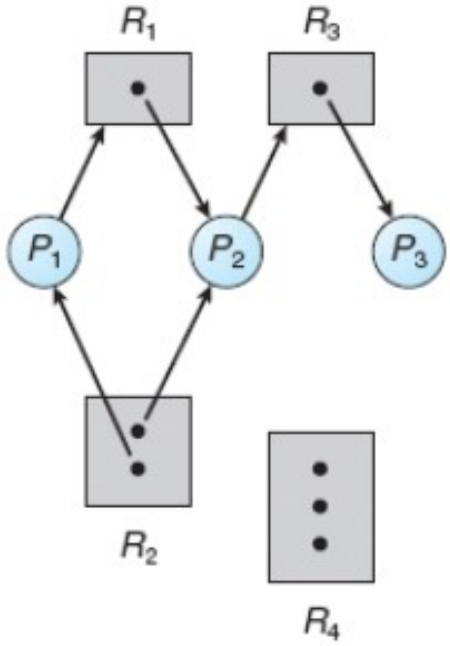
- P = {P1, P2, P3}
- R = {R1, R2, R3, R4}
- E = {P1 → R1, P2 → R3, R1 → P2, R2 → P2, R2 → P1, R3 → P3

Instâncias de recursos:

- Uma instância do tipo de recurso R1
- Duas instâncias do tipo de recurso R2
- Uma instância do tipo de recurso R3
- Três instâncias do tipo de recurso R4

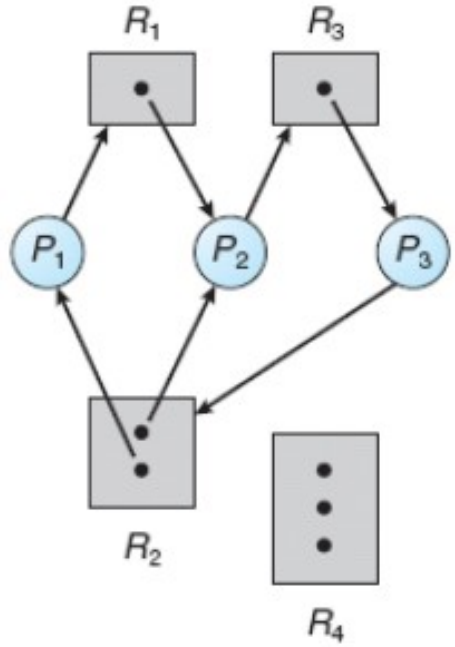
Estados dos processos:

- O processo P1 está de posse de uma instância do tipo de recurso R2 e está esperando por uma instância do tipo de recurso R1.
- O processo P2 está de posse de uma instância de R1 e de uma instância de R2 e está esperando por uma instância de R3.
- O processo P3 está de posse de uma instância de R3.



2.3 - Impasses

- **Grafo de Alocação de Recursos**



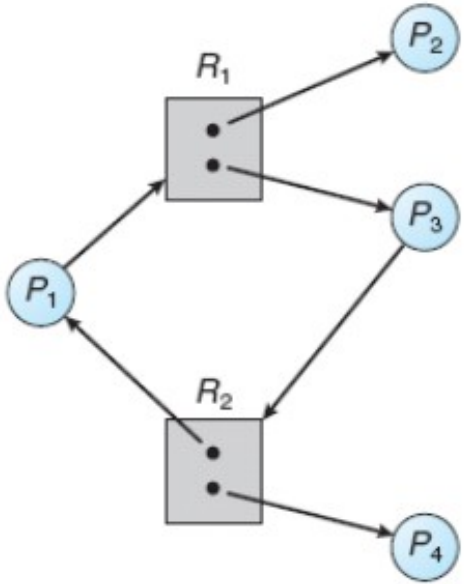
Dois ciclos

P1 → R1 → P2 → R3 → P3 → R2 → P1
P2 → R3 → P3 → R2 → P2

Os processos P1, P2 e P3 estão em deadlock.

2.3 - Impasses

- **Grafo de Alocação de Recursos**



Um ciclo

P1 → R1 → P3 → R2 → P1

Não há deadlock

2.3 - Impasses

- **Métodos para munipulação de deadlocks**

Podemos lidar com o problema do deadlock de uma entre três maneiras:

- Podemos usar um protocolo para impedir ou evitar a ocorrência de deadlocks, assegurando que o sistema **nunca** entrará em estado de deadlock.
- Podemos permitir que o sistema entre em estado de deadlock, detecteo e se recupere.
- Podemos ignorar o problema completamente e fingir que deadlocks nunca ocorrem no sistema. *

* Mais usada pela maioria dos SOs

2.3 - Impasses

- **Prevenção de deadlocks**

Ao assegurar que uma das quatro condições necessaŕis NÃO ocorra, podemos nos **prevenir** contra o deadlock:

- Exclusão Mútua
- Retenção e Espera
- Inexistência de Preempção
- Espera Circular

2.3 - Impasses

- **Prevenção de deadlocks**

Exclusão mútua

A condição de exclusão mútua deve estar presente. Isto é, pelo menos um recurso deve ser não compartilhável. Recursos compartilháveis, por outro lado, não requerem acesso mutuamente exclusivo e, portanto, não podem estar envolvidos em um deadlock. Arquivos somente de leitura são um bom exemplo de recurso compartilhável. Se vários processos tentam abrir um arquivo somente de leitura ao mesmo tempo, podem conseguir acesso simultâneo ao arquivo. Um processo nunca precisa esperar por um recurso compartilhável. Em geral, porém, não podemos prevenir a ocorrência de deadlocks negando a condição de exclusão mútua, porque alguns recursos são intrinsecamente não compartilháveis. Por exemplo, um lock mutex não pode ser compartilhado simultaneamente por vários processos.

2.3 - Impasses

- **Prevenção de deadlocks**

Retenção e Espera

Para assegurar que a condição de retenção e espera nunca ocorra no sistema, devemos garantir que, sempre que um processo solicitar um recurso, ele não esteja retendo qualquer outro recurso. Um protocolo que podemos usar requer que cada processo solicite e receba todos os seus recursos antes de começar a ser executado. Podemos implementar essa providência requerendo que as chamadas de sistema que solicitem recursos para um processo precedam todas as outras chamadas de sistema.

2.3 - Impasses

- **Prevenção de deadlocks**

Inexistência de Preempção

A terceira condição necessária para a ocorrência de deadlocks é que não haja preempção de recursos que já tenham sido alocados. Para assegurar que essa condição não ocorra, podemos usar o protocolo a seguir. Se um processo está retendo alguns recursos e solicita outro recurso que não possa ser alocado imediatamente a ele (isto é, o processo deve esperar), então todos os recursos que o processo esteja retendo no momento sofrem preempção. Em outras palavras, esses recursos são implicitamente liberados. Os recursos interceptados são adicionados à lista de recursos pelos quais o processo está esperando. O processo será reiniciado apenas quando puder reaver seus antigos recursos, assim como obter os novos que está solicitando.

2.3 - Impasses

- **Prevenção de deadlocks**

Espera Circular

Uma forma de assegurar que essa condição jamais ocorra é impor uma ordem absoluta a todos os tipos de recursos e requerer que cada processo solicite recursos em uma ordem de enumeração crescente.

Ex.: R = {R1, R2, …, Rm}

Seja uma função um-para-um F: R → N

F (drive de fita) = 1

F (drive de disco) = 5

F (impressora) = 12

2.3 - Impasses

- **Prevenção de deadlocks**

Espera Circular

1) Cada processo pode solicitar recursos apenas em uma ordem de enumeração crescente. Inicialmente um processo pode solicitar qualquer número de instâncias de um tipo de recurso — digamos, Ri.
2) O processo pode solicitar instâncias do tipo de recurso Rj se e somente se F(Rj) > F(Ri).

Alternativamente, podemos requerer que um processo que esteja solicitando uma instância do tipo de recurso Rj, tenha liberado qualquer recurso Ri de tal modo que F(Ri) ≥ F(Rj). Se várias instâncias do mesmo tipo de recurso forem necessárias, uma única solicitação deve ser emitida para todas elas.

Se esses dois protocolos forem usados, então a condição de espera circular não poderá ocorrer

2.3 - Impasses

- **Impedimento de deadlocks**

Estado Seguro

Um estado é seguro se o sistema pode alocar recursos a cada processo (até o seu máximo) em alguma ordem e continuar evitando um deadlock. Mais formalmente, um sistema está em um estado seguro somente se existe uma sequência segura.

2.3 - Impasses

- **Impedimento de deadlocks**

Estado Seguro

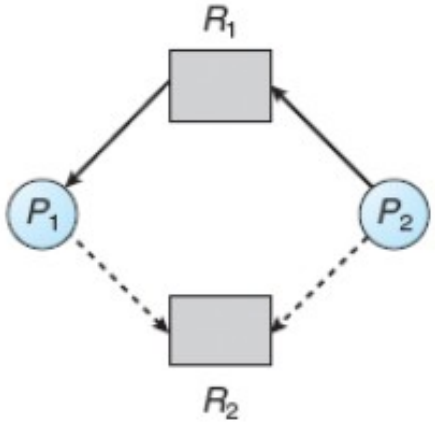
Uma sequência de processos < P1, P2, …, Pn > é uma sequência segura para o estado de alocação corrente se, para cada Pi, as solicitações de recursos que Pi ainda pode fazer podem ser atendidas pelos recursos correntemente disponíveis mais os recursos mantidos por todos os Pj, com j < i. Nessa situação, se os recursos de que Pi precisa não estiverem disponíveis imediatamente, então Pi poderá esperar até que todos os Pj tenham terminado. Quando eles estiverem encerrados, Pi poderá obter todos os recursos de que precisa, concluir sua tarefa, devolver os recursos alocados e terminar. Quando Pi terminar, Pi+1 poderá obter os recursos

2.3 - Impasses

- **Impedimento de deadlocks**

Algoritmo do Grafo de Alocação de Recursos

Uma aresta de requisição Pi → Rj indica que o processo Pi pode solicitar o recurso Rj em algum momento no futuro. Essa aresta lembra uma aresta de solicitação na direção, mas é representada no grafo por uma linha tracejada. Quando o processo Pi solicita Rj, a aresta de requisição Pi → Rj é convertida em uma aresta de solicitação. Da mesma forma, quando um recurso Rj é liberado por Pi, a aresta de atribuição Rj → Pi é reconvertida em uma aresta de requisição Pi → Rj.

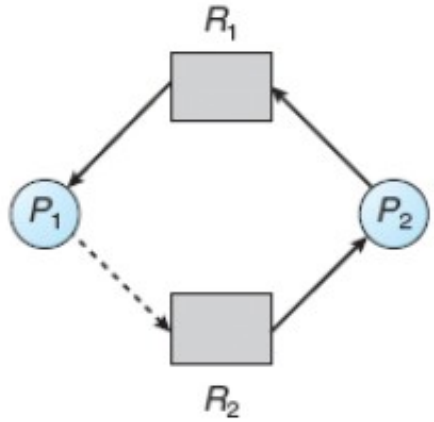


2.3 - Impasses

- **Impedimento de deadlocks**

Algoritmo do banqueiro

Quando um novo processo entra no sistema, ele deve declarar o número máximo de instâncias de cada tipo de recurso de que ele pode precisar. Esse número não pode exceder o número total de recursos no sistema. Quando um usuário solicita um conjunto de recursos, o sistema deve determinar se a alocação desses recursos o deixará em um estado seguro. Se deixar, os recursos serão alocados; caso contrário, o processo deve esperar até que algum outro processo libere recursos suficientes.



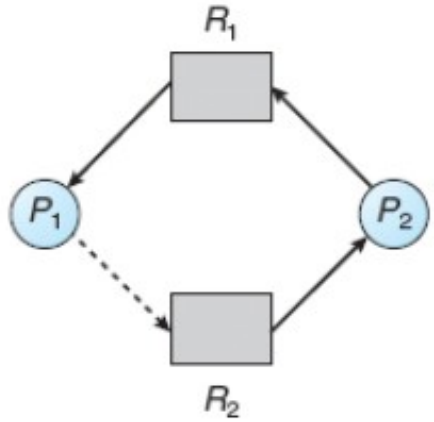
2.3 - Impasses

- **Impedimento de deadlocks**

Algoritmo do banqueiro

Precisamos de várais estruturas para implementar o algoritmo:

- **Disponível.** Um vetor de tamanho m indica o número de recursos disponíveis de cada tipo. Se **Disponível**[j] é igual a k, então k instâncias do tipo de recurso Rj estão disponíveis.
- **Max.** Uma matriz n × m define a demanda máxima de cada processo. Se **Max**[i][j] é igual a k, então, o processo Pi pode solicitar, no máximo, k instâncias do tipo de recurso Rj.
- **Alocação.** Uma matriz n × m define o número de recursos de cada tipo correntemente alocados a cada processo. Se **Alocação**[i][j] é igual a k, então o processo Pi tem correntemente alocadas k instâncias do tipo de recurso Rj.
- **Necessidade.** Uma matriz n × m indica os recursos remanescentes necessários a cada processo. Se Necessidade[i][j] é igual a k, então o processo Pi pode precisar de mais k instâncias do tipo de recurso Rj para completar sua tarefa. **Necessidade**[i][j] é igual a Max[i][j] – Alocação[i][j].



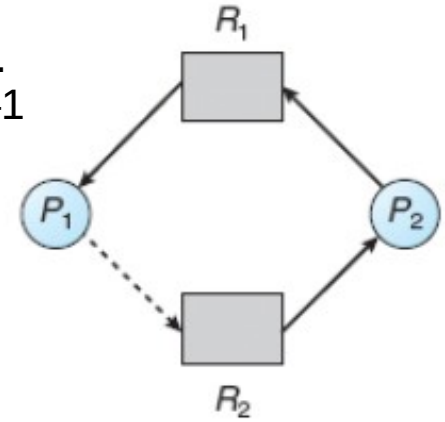
2.3 - Impasses

- **Impedimento de deadlocks**

Algoritmo de Segurança

1) Sejam Trabalho e Término vetores de tamanho m e n, respectivamente. Inicialize Trabalho = Disponível e Término[i] = false para i = 0, 1, …, n –1
2) Encontre um índice i tal que
   - a. Término[i] == false
   - b. Necessidadei ≤ Trabalho
   - Se não existir tal i, vá para o passo 4.
3) Trabalho = Trabalho + Alocaçãoi
   - Término[i] = true
   - Vá para o passo 2.
4) Se Término[i] == true para todo i, então o sistema está em um estado seguro.

Esse algoritmo pode requerer uma ordem de m × n2 operações para determinar se o estado é seguro.



2.3 - Impasses

- **Impedimento de deadlocks**

Algoritmo de Solicitação de Recursos

1. Se **Solicitaçãoi ≤ Necessidadei**, vá para o passo 2. Caso contrário, emita uma condição de erro já que o processo excedeu sua requisição máxima.
2. Se **Solicitaçãoi ≤ Disponível**, vá para o passo 3. Caso contrário, Pi deve esperar, já que os recursos não estão disponíveis.
3. Faça o sistema simular ter alocado ao processo Pi os recursos solicitados modificando o estado, como descrito abaixo:

Disponível = Disponível – Solicitaçãoi;
Alocaçãoi = Alocaçãoi + Solicitaçãoi;
Necessidadei = Necessidadei – Solicitaçãoi.

Se o estado de alocação de recursos resultante é seguro, a transação é concluída e o processo Pi recebe seus recursos. No entanto, se o novo estado é inseguro, então Pi deve esperar por **Solicitaçãoi**, e o estado de alocação de recursos anterior é restaurado.

```python
import threading
import random
import time

#inheriting threading class in Thread module
class Philosopher(threading.Thread):
    running = True  #used to check if everyone is finished eating

 #Since the subclass overrides the constructor, it must make sure to invoke the base class constructor (Thread.__init__()) before doing anything else to the thread.
    def __init__(self, index, forkOnLeft, forkOnRight):
        threading.Thread.__init__(self)
        self.index = index
        self.forkOnLeft = forkOnLeft
        self.forkOnRight = forkOnRight

         for i in range(5)]

    Philosopher.running = True
    for p in philosophers: p.start()
    time.sleep(100)
    Philosopher.running = False
    print ("Now we're finishing.")


if __name__ == "__main__":
    main()


    def run(self):
        while(self.running):
            # Philosopher is thinking (but really is sleeping).
            time.sleep(30)
            print ('Philosopher %s is hungry.' % self.index)
            self.dine()

    def dine(self):
        # if both the semaphores(forks) are free, then philosopher will eat
        fork1, fork2 = self.forkOnLeft, self.forkOnRight
        while self.running:
            fork1.acquire() # wait operation on left fork
            locked = fork2.acquire(False)
            if locked: break #if right fork is not available leave left fork
            fork1.release()
            print ('Philosopher %s swaps forks.' % self.index)
            fork1, fork2 = fork2, fork1
        else:
            return
        self.dining()
        #release both the fork after dining
        fork2.release()
        fork1.release()

    def dining(self):
        print ('Philosopher %s starts eating. '% self.index)
        time.sleep(30)
        print ('Philosopher %s finishes eating and leaves to think.' % self.index)

def main():
    forks = [threading.Semaphore() for n in range(5)] #initialising array of semaphore i.e forks

    #here (i+1)%5 is used to get right and left forks circularly between 1-5
    philosophers= [Philosopher(i, forks[i%5], forks[(i+1)%5])
```

```
> python3 diningphilosophers.py
Philosopher 0 is hungry.
Philosopher 0 starts eating.
Philosopher 1 is hungry.
Philosopher 2 is hungry.
Philosopher 2 starts eating.
Philosopher 3 is hungry.
Philosopher 4 is hungry.
Philosopher 4 swaps forks.
Philosopher 0 finishes eating and leaves to think.
Philosopher 1 swaps forks.
Philosopher 2 finishes eating and leaves to think.
Philosopher 4 starts eating.
Philosopher 3 swaps forks.
Philosopher 1 starts eating.
Philosopher 0 is hungry.
Philosopher 2 is hungry.
Philosopher 4 finishes eating and leaves to think.
Philosopher 1 finishes eating and leaves to think.
Philosopher 0 swaps forks.
Philosopher 0 starts eating.
Philosopher 2 starts eating.
Philosopher 3 swaps forks.
Now we're finishing.
Philosopher 4 is hungry.
Philosopher 1 is hungry.
Philosopher 2 finishes eating and leaves to think.
Philosopher 0 finishes eating and leaves to think.
Philosopher 3 starts eating.
Philosopher 3 finishes eating and leaves to think.x
```

Contato:

“wagner.costa@cest.edu.br”