Deadlocks

Sistemas Operacionais
2017-1
Flavio Figueiredo (http://flaviovdf.github.io)

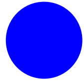
Deadlock

Conjunto de processos bloqueados impedidos progredir

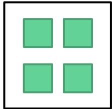
Modelo: grafo de alocação de recursos

Modelo: grafo de alocação de recursos

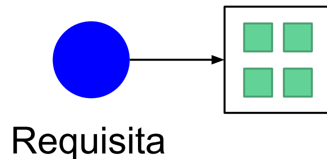
- Processo

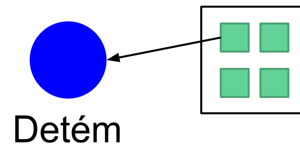


- Recurso com quatro instâncias

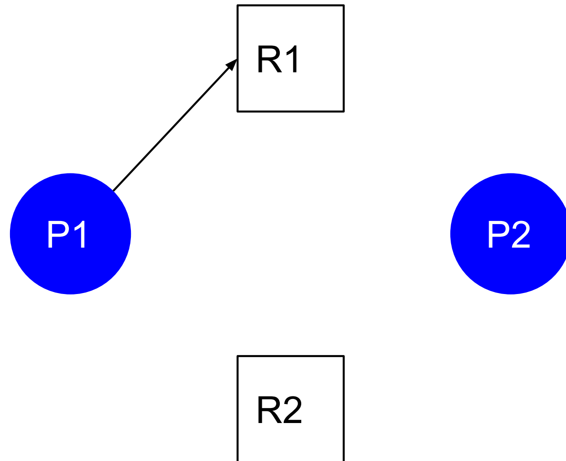


- Pi requisita instância de Rj, aresta saída do processo (esquerda)
- Pi detém uma instância de Rj, aresta de entrada no processo (direita)

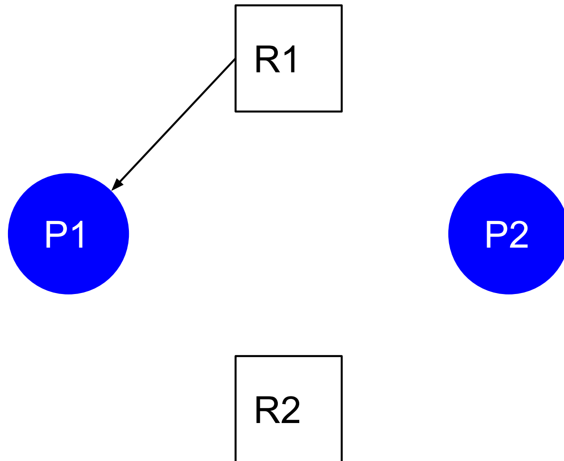




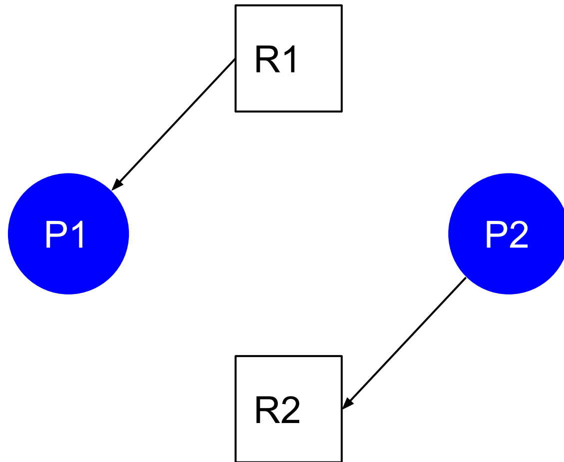
Exemplo: Apenas 1 recurso de cada tipo



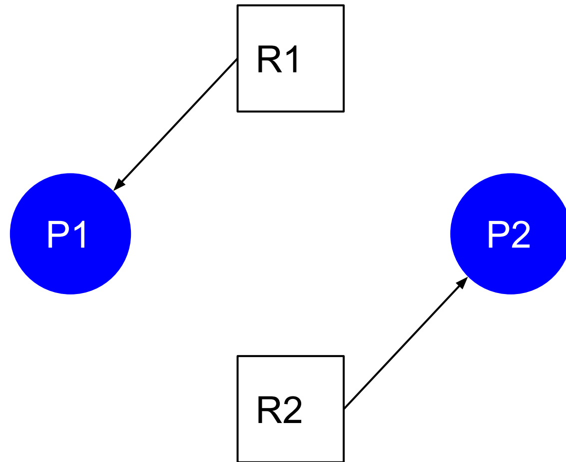
Exemplo



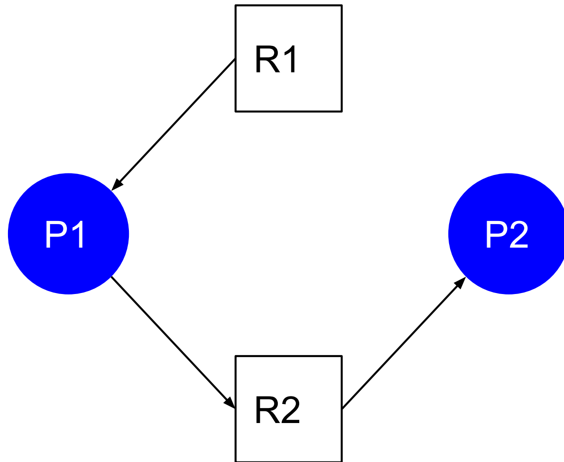
Exemplo



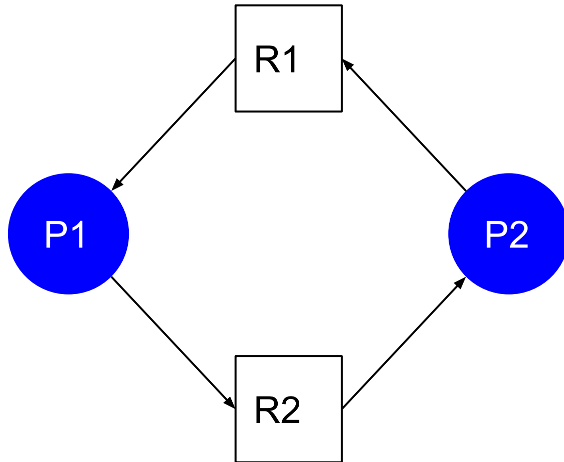
Exemplo



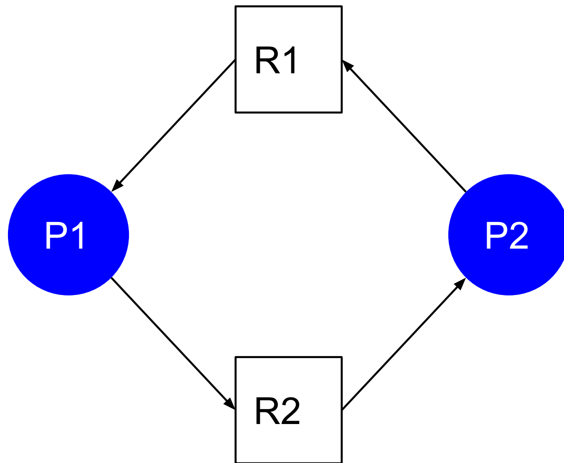
Exemplo



Exemplo



Exemplo



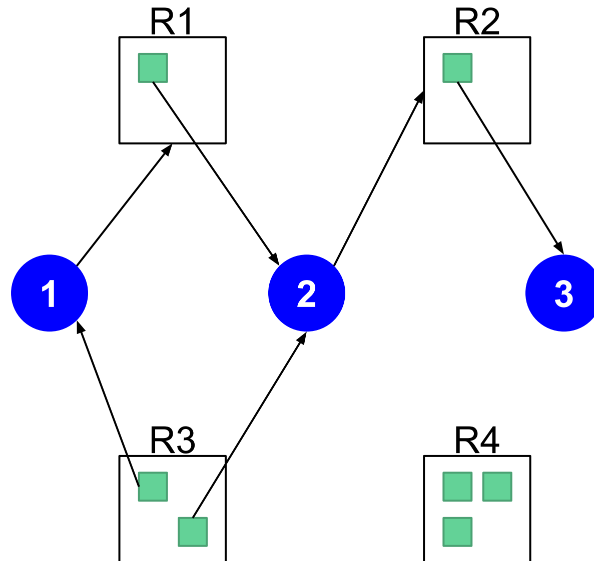
DEADLOCK

Ciclos e Deadlocks

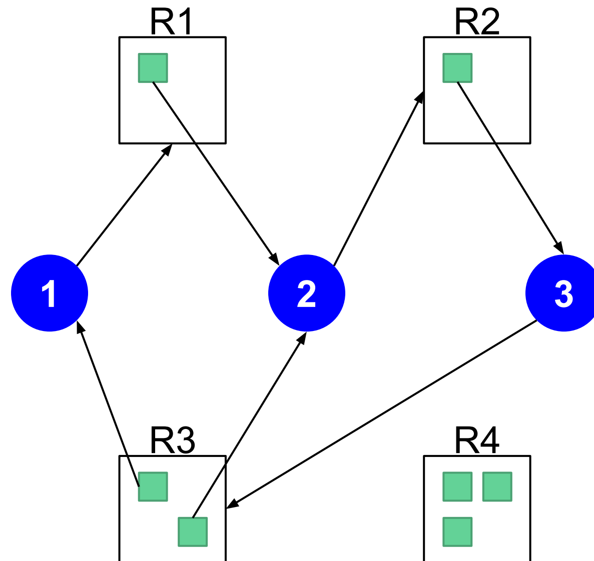
- Se não há ciclos do grafo ➔ não há deadlock

- Se há ciclos no grafo
  - Se recursos só têm uma instância ➔ deadlock
  - Se recursos têm múltiplas instâncias ➔ possível deadlock

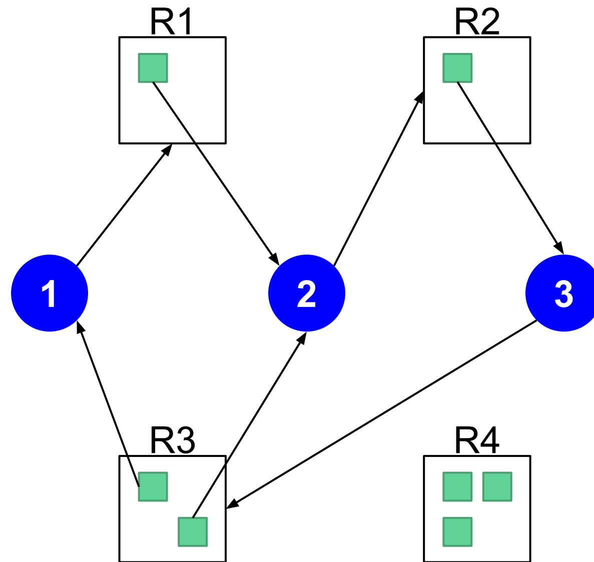
Detecção de deadlock quando recursos têm múltiplas instâncias



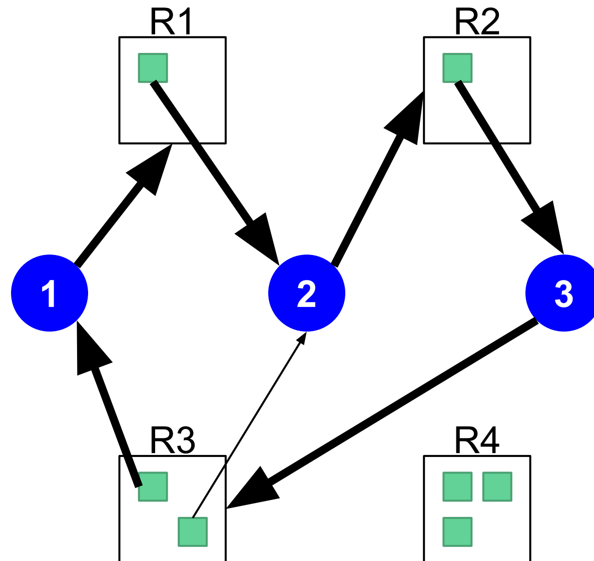
P3 Solicita R3



Deadlock?!

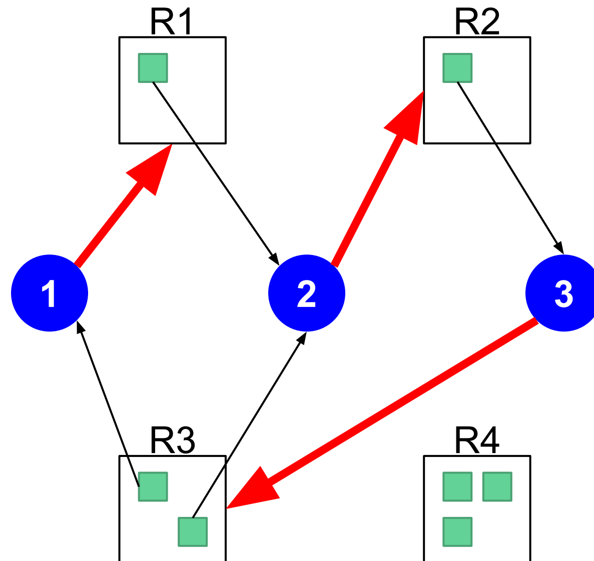


Temos o Ciclo

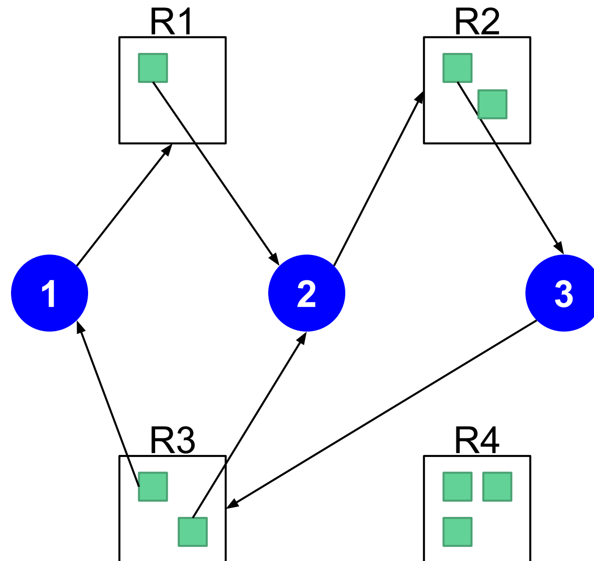


Deadlock?!

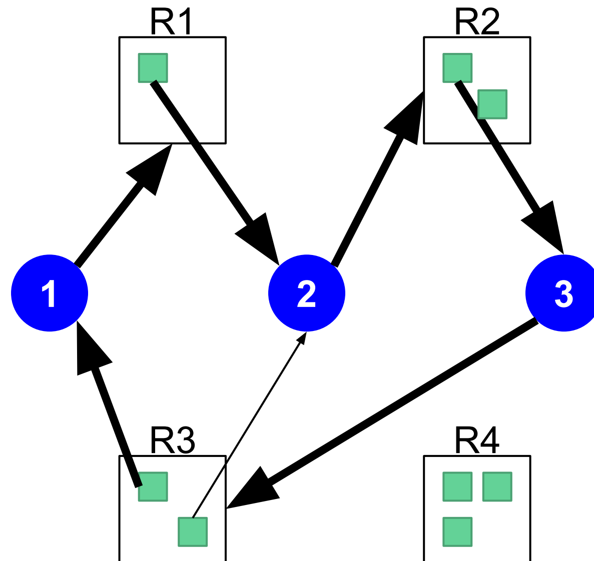
Sim. Ninguém consegue fazer mais nada



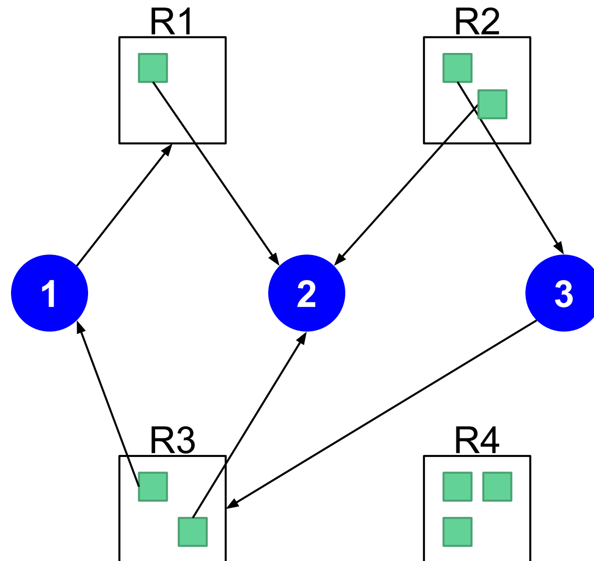
E se tiver mais recursos R2?



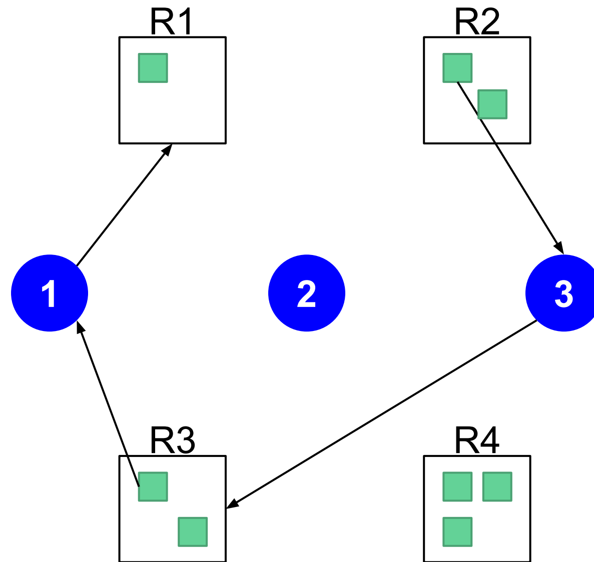
E se tiver mais recursos R2?
Ainda temos ciclo



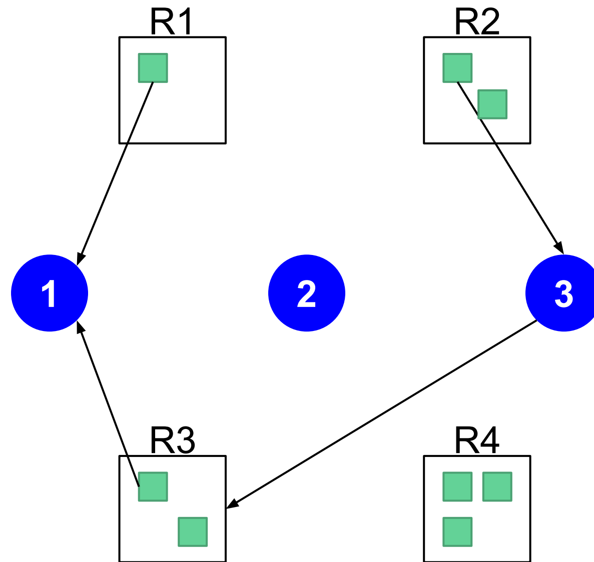
Porém, P2 consegue um recurso



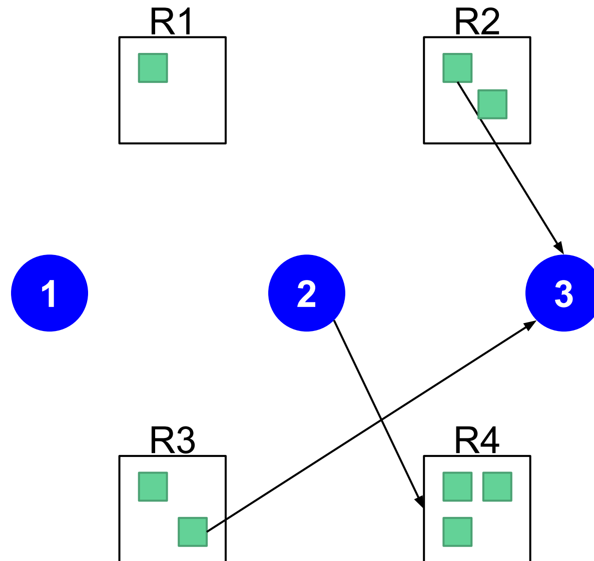
P2 pode liberar recursos



Vida Continua

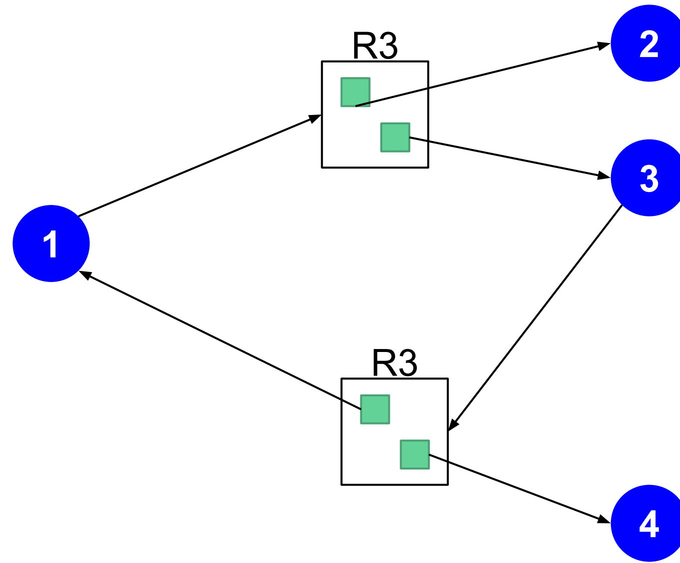


Vida Continua

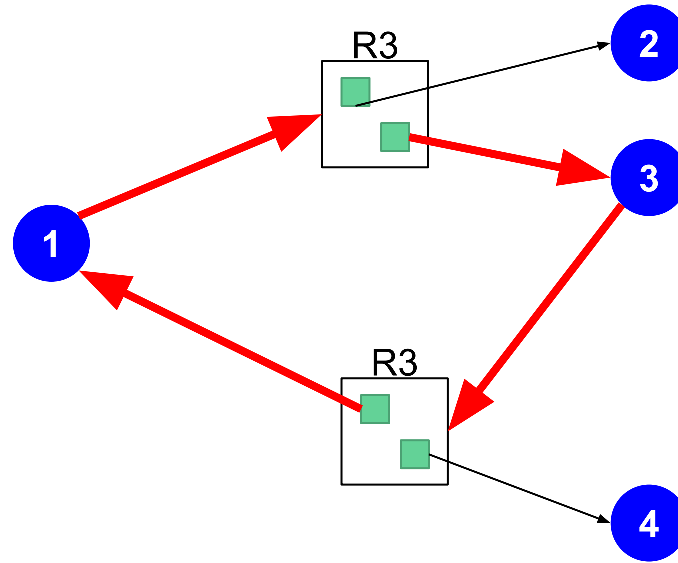


[V ou F] Se todas as instâncias de todos os recursos estiverem ocupadas e tivermos ciclo ➜ Deadlock?

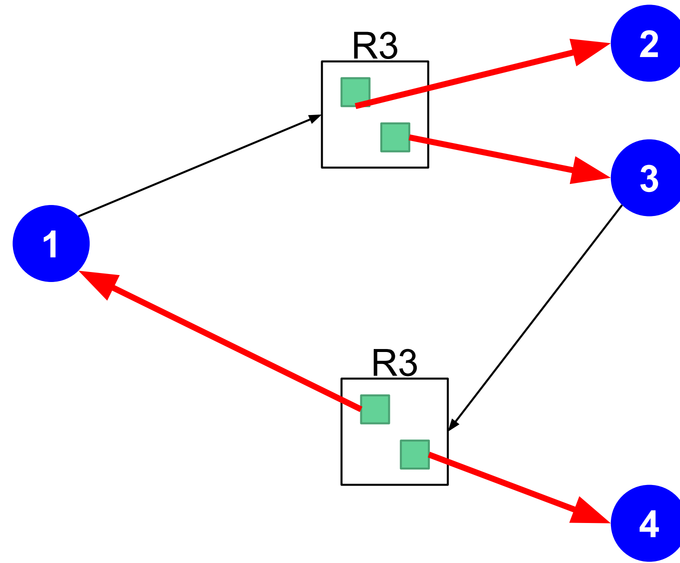
[Contra Exemplo] Caso **sem** deadlock quando recursos têm múltiplas instâncias.



[Contra Exemplo] Caso **sem** deadlock quando recursos têm múltiplas instâncias.

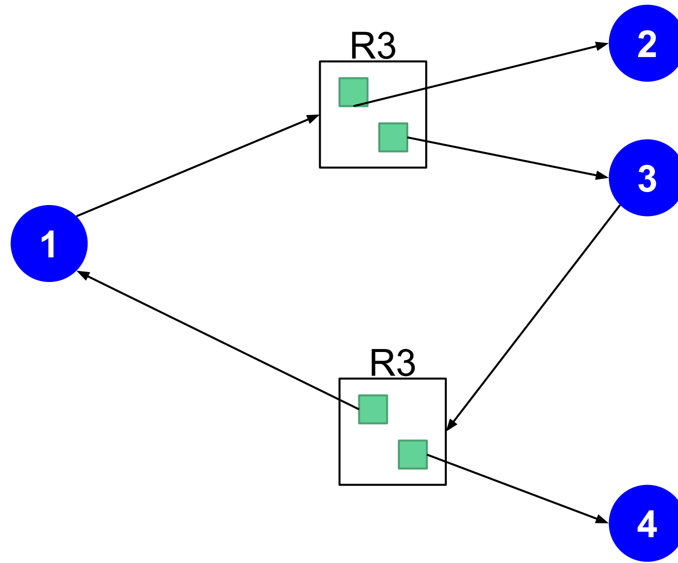


[Contra Exemplo] Caso **sem** deadlock quando recursos têm múltiplas instâncias.

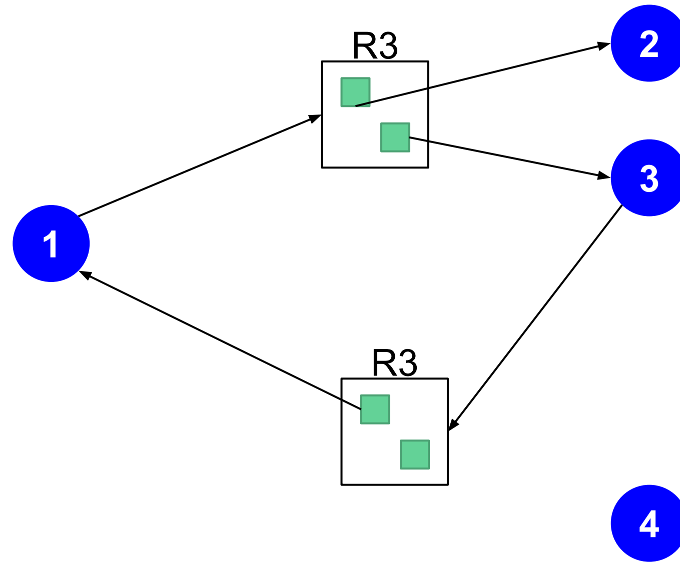


[Contra Exemplo] Caso **sem** deadlock quando recursos têm múltiplas instâncias.

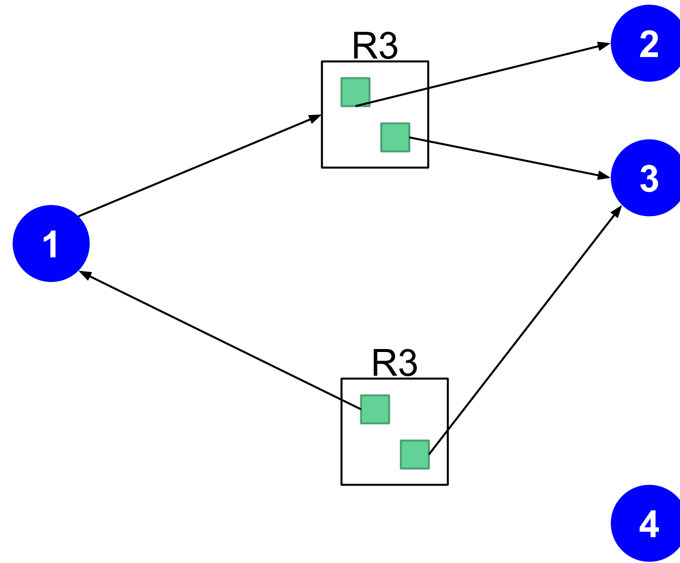
4 está de boa
2 está de boa



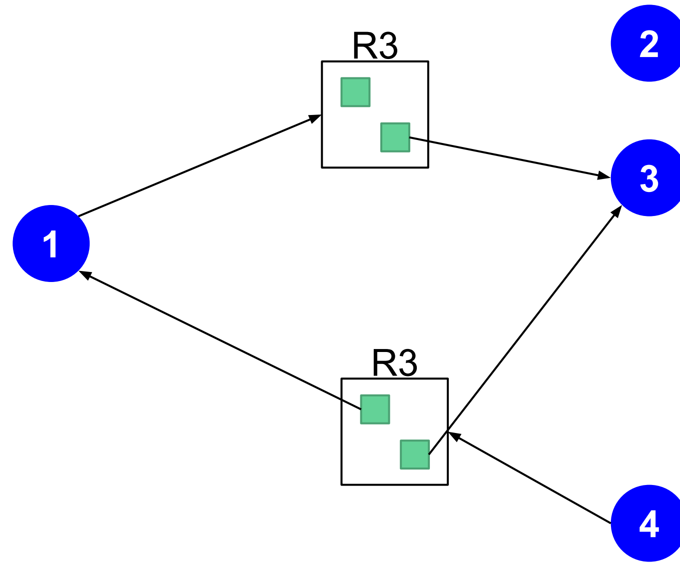
Vida Continua



Vida Continua



Vida Continua



Condições necessárias

1. Exclusão mútua
   a. Lembrar de sincronização
   b. Apenas 1 processo por vez na região crítica

2. Posse e espera
   a. Todos os processos envolvidos estão esperando algum recurso
   b. Não existem recursos livres para os processos em espera
      i. Alguém tem posse deles
   c. No exemplo anterior, P2 e P4 não esperavam por nada

3. Não preempção do recurso
   a. Não temos como liberar um recurso **na marra**

4. Espera circular

Formas de lidar com deadlock

- Prevenir, por construção, que deadlocks aconteçam
- Impedir deadlocks antes que ocorram
- Detectar e corrigir deadlocks quando acontecerem
- Ignorar deadlocks

Prevenção de deadlocks

Prevenção de deadlocks

Impedir que alguma das condições necessárias ocorra:

1. Exclusão mútua
2. Posse e espera
3. Não preempção do recurso
4. Espera circular

1. Eliminar exclusão mútua

1. Eliminar exclusão mútua

- Não é uma opção, sorry.

2. Eliminar posse e espera

Impedir que processos requisitem recursos aos poucos

- Pedir todos recursos de uma vez
- Liberar todos os recursos antes de pedir novos

Pode levar a baixa utilização dos recursos ou inanição

2. Eliminar posse e espera

Impedir que processos requisitem recursos aos poucos

- Pedir todos recursos de uma vez
- Liberar todos os recursos antes de pedir novos

Pode levar a baixa utilização dos recursos ou inanição

Solução do Garçom nos Filósofos

3. Preemptar recursos

Permitir que o sistema recupere recurso alocado a um processo “a força”

- Quando um processo tentar alocar um recurso e não conseguir ele libera os recursos que havia alocado
    - Recursos liberados entram na lista de requisição do processo

Aplicável a recursos cujo estado pode ser facilmente restaurado

3. Preemptar recursos

Permitir que o sistema recupere recurso alocado a um processo “a força”

- Quando um processo tentar alocar um recurso e não conseguir ele libera os recursos que havia alocado
    - Recursos liberados entram na lista de requisição do processo

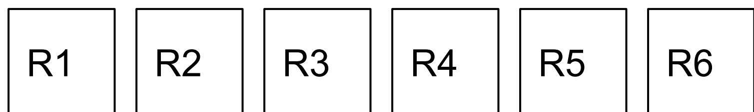
Aplicável a recursos cujo estado pode ser facilmente restaurado

Se um deadlock está sendo causado por um processo tentando imprimir, podemos preemptar o mesmo. O usuário vai ver que não imprimiu e tentará novamente.

4. Eliminar espera circular

Definir uma ordem total para os recursos disponíveis

- Cada processo deve requisitar recursos em ordem crescente
- Impede formação de ciclo no grafo



4. Eliminar espera circular

```
void xfer(Account from, Account to, double amount) {
    Mutex m1, m2;
    m1 = getlock(from);
    m2 = getlock(to);
    wait(m1);
    wait(m2);
    withdraw(from, amount);
    deposit(to, amount);
    signal(m2);
    signal(m1);
}
```

4. Eliminar espera circular

```
void xfer(Account from, Account to, double amount) {
    Mutex m1, m2;
    m1 = getlock(from);
    m2 = getlock(to);
    wait(m1);
    wait(m2);
    withdraw(from, amount);
    deposit(to, amount);
    signal(m2);
    signal(m1);
}
```

```
// Bob no Caixa Eletrônico
xfer(accountBob, accountJane, amountA)

// Ao mesmo tempo, Jane no Banco SO
xfer(accountJane, accountBob, amountB)
```

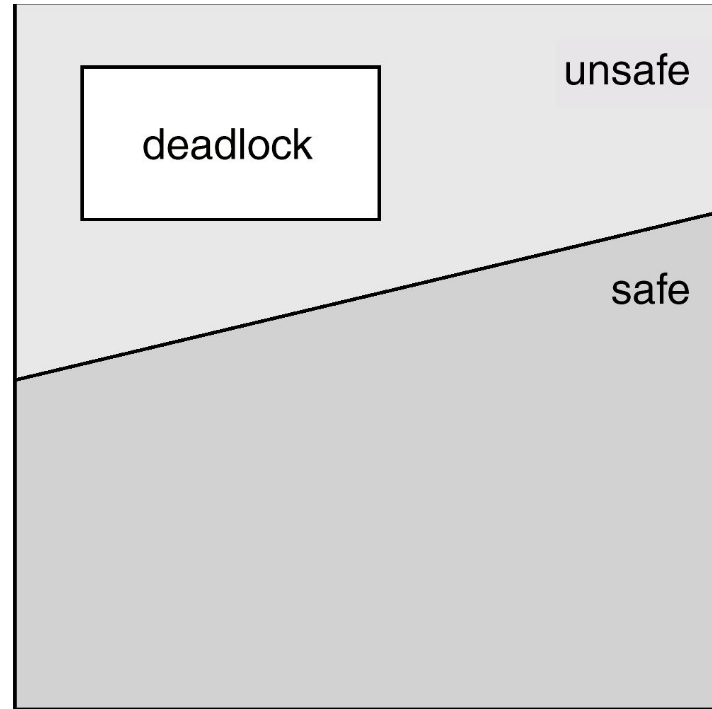
4. Eliminar espera circular

```
void xfer(Account from, Account to, double amount) {
    Mutex m1, m2;
    if (from.ID < to.ID) {
     m1 = getlock(from);
     m2 = getlock(to);
    }
    else {
     m1 = getlock(to);
     m2 = getlock(from);
    }
    wait(m1);
    wait(m2);
    withdraw(from, amount);
    deposit(to, amount);
    signal(m2);
    signal(m1);
}
```

E se tivermos + de 2 recursos?

Impedimento de deadlocks

Estado seguro vs deadlocks



Impedimento de deadlocks

- Cada processo declara necessidade máxima de recursos
- Avaliar dinamicamente cada alocação

Alocação é feita para *garantir* que não haverá espera circular

Requisitos do sistema

- Pi pode esperar o término de outros processos
  - Mesmo se os recursos estiverem disponíveis
- Quando Pi termina, ele libera todos os recursos

Estado seguro

Informalmente: os recursos que *cada* Pi pode requisitar não excedem a soma do que está disponível mais o que *será* liberado por outros processos

Formalmente: é necessário existir uma sequência de término dos processos assumindo que cada processo pode requisitar até sua necessidade máxima

Estado seguro - exemplo

12 acionadores de fita (recursos), 3 processos

	P1	P2	P3
Necessidade máxima	10	4	9
Necessidade atual	5	2	2

Algoritmo do grafo de alocação de recursos

Válido quando recursos têm apenas uma instância

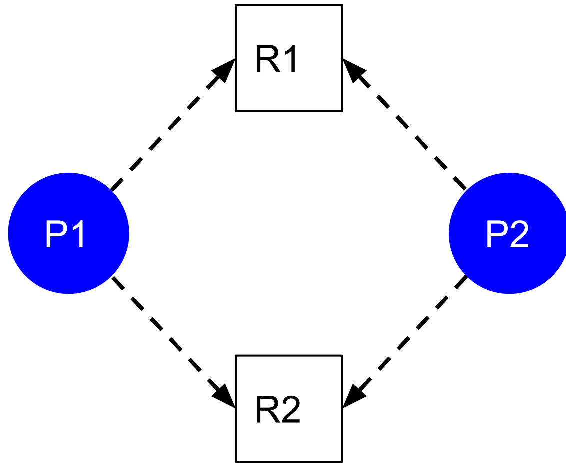
Aresta de requisição Pi → Rj (tracejada)

- Indica que processo Pi pode requisitar Rj
- Declaradas *a priori*

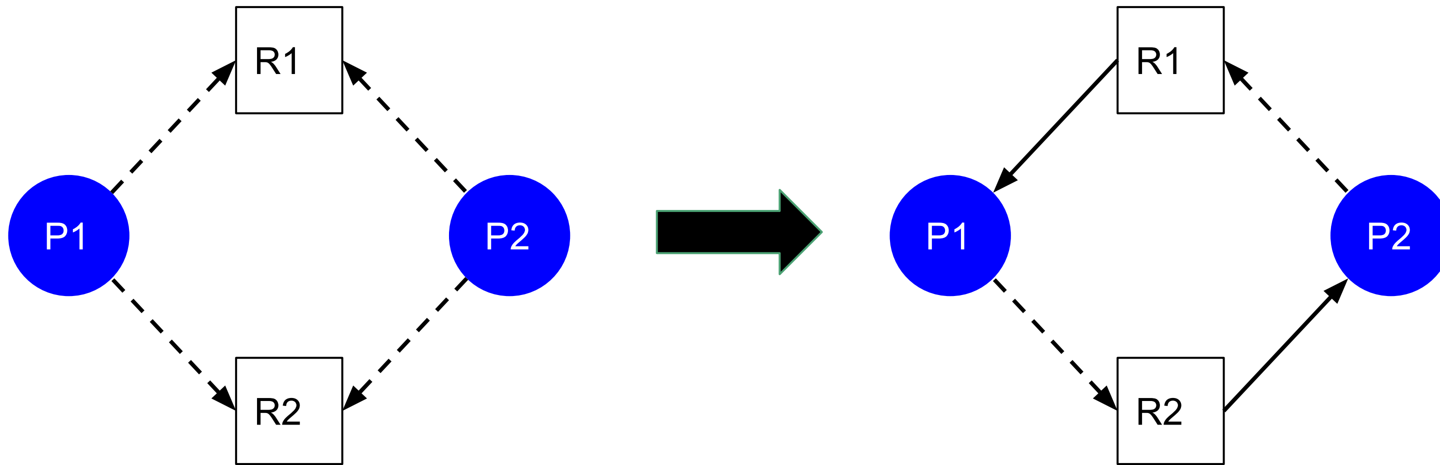
Antes de atender uma requisição, verificar se cria um ciclo

- Se criar ciclo (com arestas sólidas e tracejadas) o estado seria inseguro

Exemplo



Exemplo



Algoritmo do banqueiro

Aplicável a recursos com múltiplas instâncias

Para N processos e M tipos de recursos, mantemos as seguintes variáveis:

- disponível[m]: número de instâncias disponíveis de cada recurso
- max[n][m]: demanda máxima de cada processo
- alocação[n][m]: alocação dos recursos entre os processos
- necessidade[n][m]: quando cada processo ainda pode requisitar

Algoritmo do banqueiro

```
inicialização:                                    #Assumindo que as matrizes são globais
   acessível = disponível
   término[n] = [0 para todos os processos]
verifica():
   estado_mudou = True
   while estado_mudou:
        estado_mudou = False
        para cada processo i de 0 a N-1:
            se termino[i] == 0 e all(necessidade[i] < acessível[i]):
                libera(i)
                estado_mudou = True
   final()
libera(i):
    acessível += alocação[i]
    termino[i] = 1
final():
    seguro = min(término)
```

Algoritmo do banqueiro

```
inicialização():
   acessível = disponível
   término[n] = [0 para todos os processos]
verifica():
   estado_mudou = True
   while estado_mudou:
        estado_mudou = False
        para cada processo i de 0 a N-1:
            se termino[i] == 0 e all(necessidade[i] < acessível[i]):
                libera(i)
                estado_mudou = True
   final()
libera(i):
    acessível += alocação[i]
    termino[i] = 1
final():
    seguro = min(término)
```

- **Ainda não sabemos se o processo pode terminar**
- **all → verifica se tudo é true no vetor**
- **Porém, se tem recursos recursos para terminar**
  - **Mudar estado**

Algoritmo do banqueiro

```
inicialização():
   acessível = disponível
   término[n] = [0 para todos os processos]
verifica():
   estado_mudou = True
   while estado_mudou:
        estado_mudou = False
        para cada processo i de 0 a N-1:
            se termino[i] == 0 e all(necessidade[i] < acessível[i]):
                libera(i)
                estado_mudou = True
   final()
libera(i):
    acessível += alocação[i]
    termino[i] = 1
final():
    seguro = min(término)
```

- **Como o processo pode terminar**
  - **Liberamos os recursos (eventually)**
  - **Mais chances para os outros**

Algoritmo do banqueiro

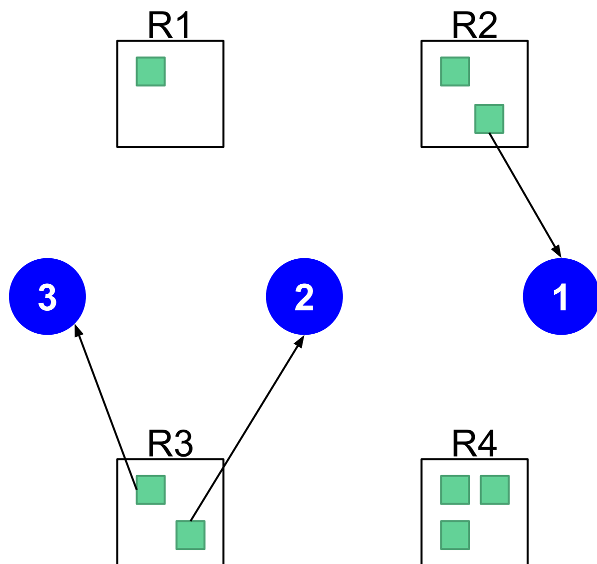
```
inicialização():
   acessível = disponível
   término[n] = [0 para todos os processos]
verifica():
   estado_mudou = True
   while estado_mudou:
        estado_mudou = False
        para cada processo i de 0 a N-1:
            se termino[i] == 0 e all(necessidade[i] < acessível[i]):
                libera(i)
                estado_mudou = True
   final()
libera(i):
    acessível += alocação[i]
    termino[i] = 1
final():
    seguro = min(término)
```

- Se algum 0, então alguém não termina
- Unsafe!

Usando o algoritmo do banqueiro

```
processo n envia solicitação[m]:                     #Podemos fazer múltiplos ao mesmo tempo
   se solicitação > necessidade[n]: ERRO
   se solicitação > disponível: AGUARDA
   "aloca" recursos para satisfazer solicitação    #Como a vida seria com essa alocação?
   se estado é seguro: ALOCADO                       #Roda algoritmo do banqueiro
   caso contrário: AGUARDA
```

Banqueiro Exemplo



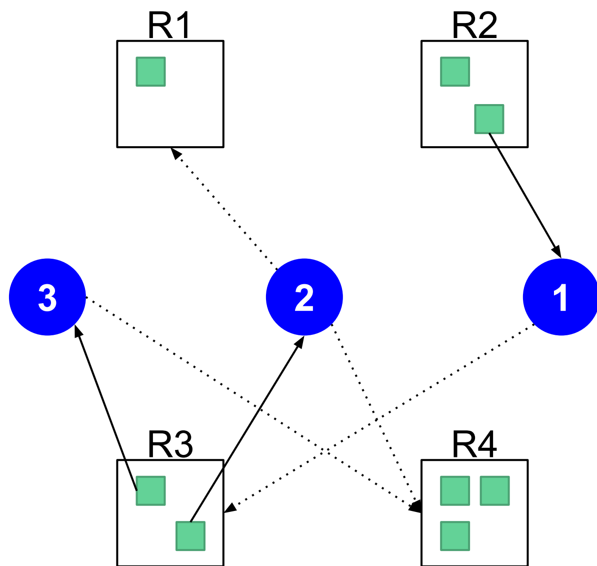
Disponível			
R1	R2	R3	R4
1	1	0	3

Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1				
P2				
P3				

- Total podemos ver na figura
- Assumindo Max = Total (todo mundo pode pedir tudo)

Rodando o While



Disponível			
R1	R2	R3	R4
1	1	0	3

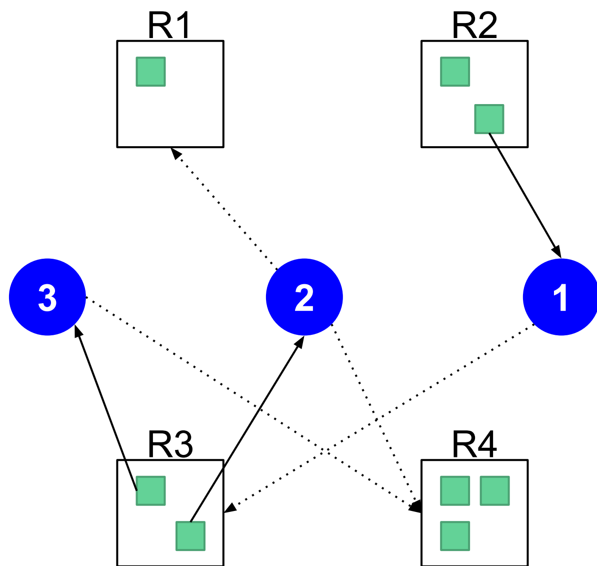
Acessível			
R1	R2	R3	R4
1	1	0	3

Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	1	0
P2	1	0	0	1
P3	0	0	0	1

```
P1
    se termino[i] == 0 e all(necessidade[i] < acessível[i])
```

P1 não entra no IF



Disponível			
R1	R2	R3	R4
1	1	0	3

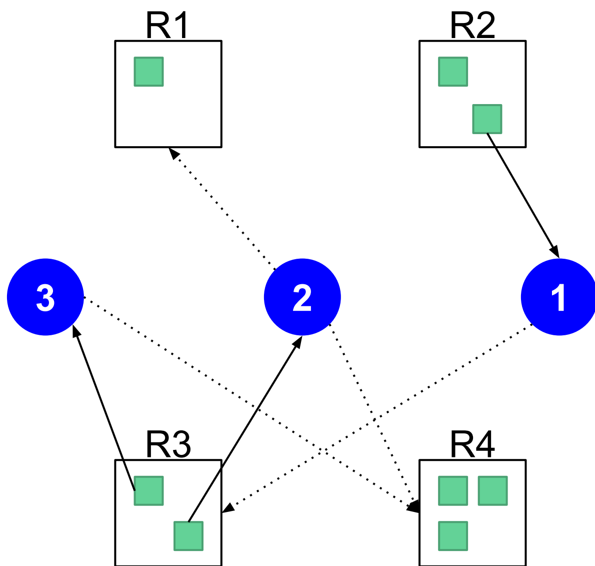
Acessível			
R1	R2	R3	R4
1	1	0	3

Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	1	0
P2	1	0	0	1
P3	0	0	0	1

```
P1
    se termino[i] == 0 e all(necessidade[i] < acessível[i])
        FALSE
```

P2 OK



Disponível			
R1	R2	R3	R4
1	1	0	3

Acessível			
R1	R2	R3	R4
1	1	1	3

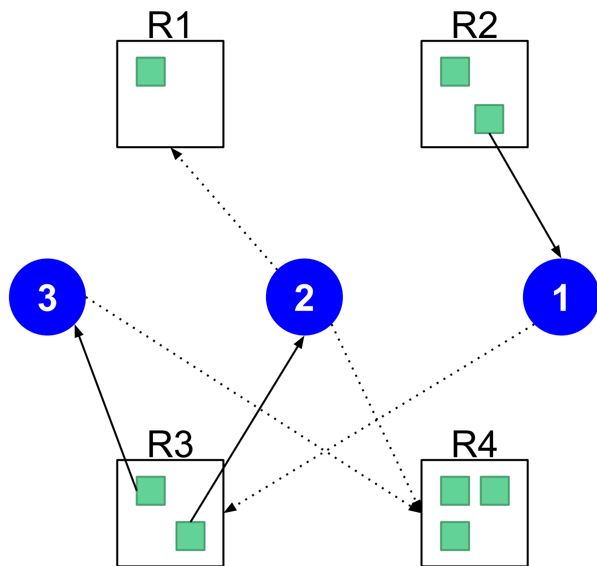
Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	1	0
P2	1	0	0	1
P3	0	0	0	1

P2

```
se termino[i] == 0 e all(necessidade[i] < acessível[i])
    TRUE
    acessivel[i] += alocacao[i]
    termino[i] += 1
```

P3 OK. Voltamos no While



Disponível			
R1	R2	R3	R4
1	1	0	3

Acessível			
R1	R2	R3	R4
1	1	2	3

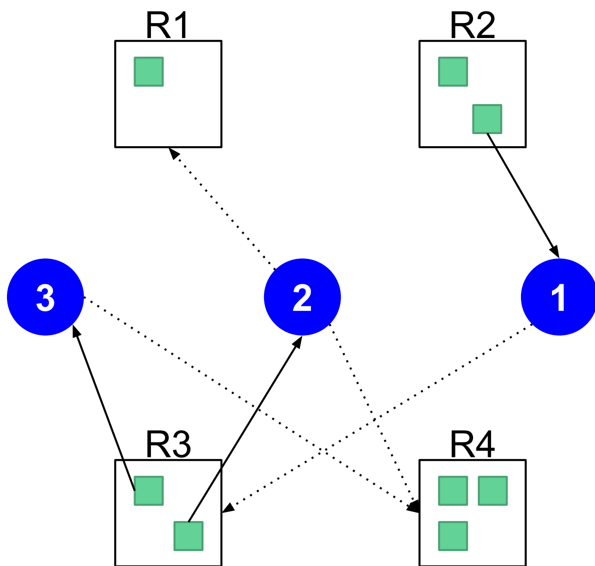
Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	1	0
P2	1	0	0	1
P3	0	0	0	1

P3

```
se termino[i] == 0 e all(necessidade[i] < acessível[i])
    TRUE
    acessivel[i] += alocacao[i]
    termino[i] += 1
```

P1 OK Agora!



Disponível			
R1	R2	R3	R4
1	1	0	3

Acessível			
R1	R2	R3	R4
1	1	2	3

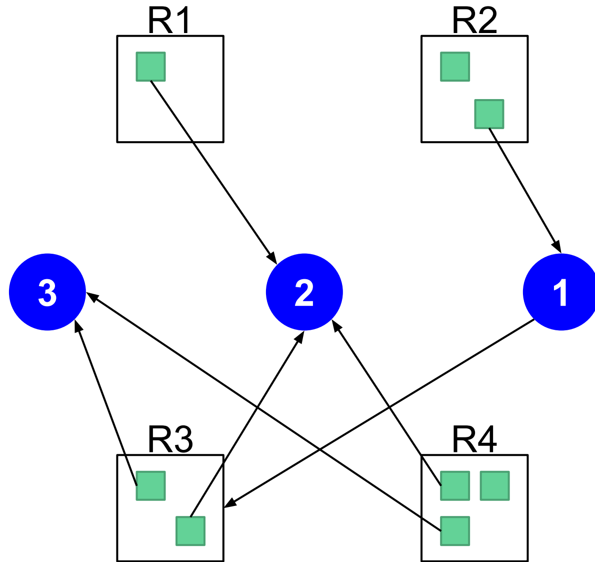
Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	1	0
P2	1	0	0	1
P3	0	0	0	1

P1

```
se termino[i] == 0 e all(necessidade[i] < acessível[i])
    TRUE
    acessivel[i] += alocacao[i]
    termino[i] += 1
```

Fim (após receber recursos)

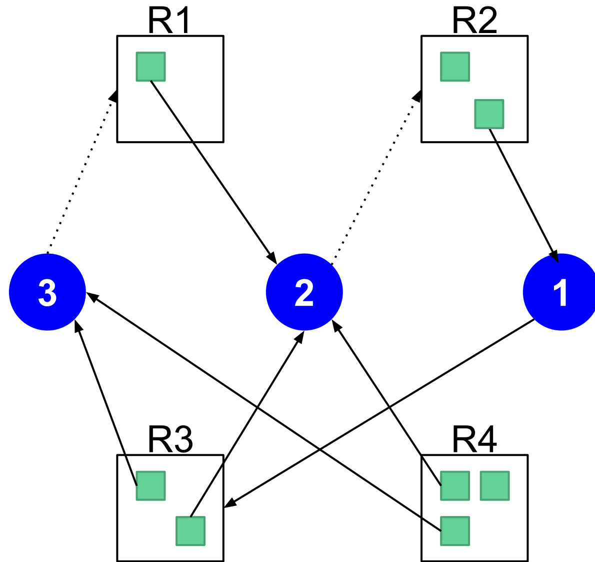


Disponível			
R1	R2	R3	R4
0	1	0	1

Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	1
P3	0	0	1	1

Necessidade				
	R1	R2	R3	R4
P1				
P2				
P3				

Outra Rodada



Disponível			
R1	R2	R3	R4
0	1	0	1

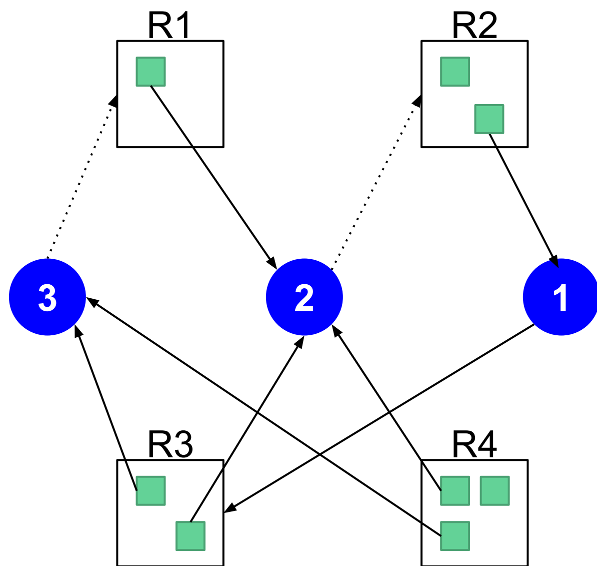
Acessível			
R1	R2	R3	R4
0	1	0	1

Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	1
P3	0	0	1	1

Necessidade				
	R1	R2	R3	R4
P1	0	0	0	0
P2	0	1	0	0
P3	1	0	0	0

P1 não entra na brincadeira. Não requisitou nada

Outra Rodada



Disponível			
R1	R2	R3	R4
0	1	0	1

Acessível			
R1	R2	R3	R4
1	1	0	1

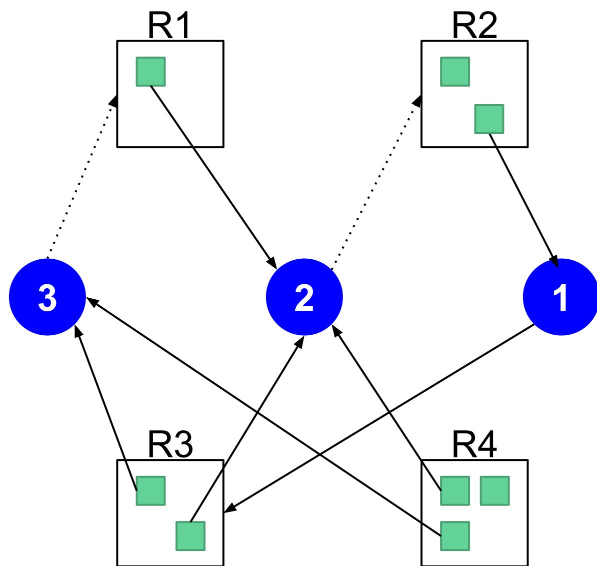
Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	0	0
P2	0	1	0	0
P3	1	0	0	0

P2

```
    se termino[i] == 0 e all(necessidade[i] < acessível[i])
        TRUE
        acessivel[i] += alocacao[i]
        termino[i] += 1
```

Outra Rodada



Disponível			
R1	R2	R3	R4
0	1	0	1

Acessível			
R1	R2	R3	R4
1	1	1	2

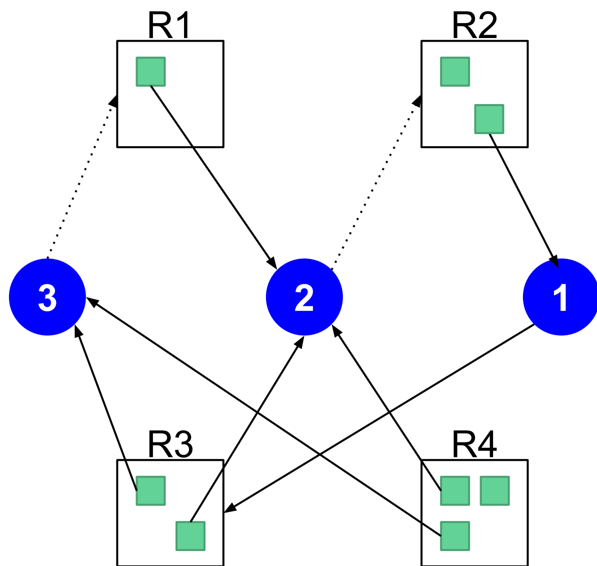
Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	0	0
P2	0	1	0	0
P3	1	0	0	0

P3

```
    se termino[i] == 0 e all(necessidade[i] < acessível[i])
        TRUE
        acessivel[i] += alocacao[i]
        termino[i] += 1
```

Outra Rodada



Disponível			
R1	R2	R3	R4
0	1	0	1

Acessível			
R1	R2	R3	R4
1	1	1	2

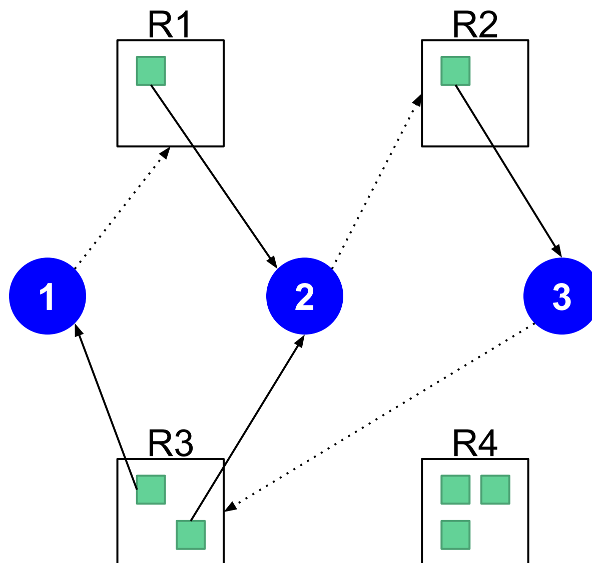
Alocação				
	R1	R2	R3	R4
P1	0	1	0	0
P2	1	0	1	0
P3	0	0	1	0

Necessidade				
	R1	R2	R3	R4
P1	0	0	0	0
P2	0	1	0	0
P3	1	0	0	0

Temos um possível laço mas estamos seguro!
R2 suficiente!

Caso com Deadlock

- Ninguém entra no IF
- Seguro
  - False!!



Detecção de deadlocks

Detecção de deadlocks

Deixa o sistema entrar o deadlock

Detectar o deadlock (semelhante ao algoritmo de impedimento)

Defina um esquema de recuperação

Detecção de deadlock quando recursos têm apenas uma instância

Deteção de ciclo em grafos $O(n^2)$

Detecção de deadlock quando recursos têm múltiplas instâncias

Algoritmo semelhante ao algoritmo do banqueiro

Para N processos e M tipos de recursos, mantemos as seguintes variáveis:

- disponível[m]: número de instâncias disponíveis de cada recurso
- alocação[n][m]: alocação dos recursos entre os processos
- solicitação[n][m]: quando cada processo requisitou

Detecção de deadlock quando recursos têm múltiplas instâncias

```
inicialização:                                    #Assumindo que as matrizes são globais
   acessível = disponível
   término[n] = [0 para todos os processos]
verifica():
   estado_mudou = False
   while estado_mudou:
        estado_mudou = False
        para cada processo i de 0 a N-1:
            se termino[i] == 0 e all(necessidade[i] < acessível[i]):
                libera(i)
                estado_mudou = True
   final()
libera(i):
    acessível += alocação[i]
    termino[i] = 1
final():
    deadlock = min(término) == 0  #Alguém não finaliza
```

Aplicação do algoritmo de detecção

Quão frequentes são deadlocks

Quantidade de processos envolvidos

Tempo de espera

Sobrecarga

Resolução de deadlocks

É preciso quebrar a dependência circular entre os processos

Abortar um ou mais processos: terminação abrupta e estado inconsistente

Preempção de recursos: restauração de estado prévio

Decisões de projeto:

- Qual processo cancelar?
- Redistribuição dos recursos?
- Inanição

Casos de Uso

Casos de Uso

- ???

Casos de Uso

- ???
  - Realmente não conheço SO moderno que utiliza os algoritmos acima
  - Melhor fazer código que não entra em deadlock

Casos de Uso

- ???
  - Realmente não conheço SO moderno que utiliza os algoritmos acima
  - Melhor fazer código que não entra em deadlock
- De qualquer forma
  - Conhecimento útil para teste e desenvolvimento de sistemas
    - https://github.com/sasha-s/go-deadlock
    - https://github.com/golang/go/issues/13759
    - https://www.yourkit.com/docs/java/help/deadlocks.jsp