Sistemas Operacionais: Deadlocks

Prof. Jose Macedo
Sistemas Operacionais

Objetivos Aula

- Descrever os deadlocks, os quais impedem que vários processos concorrentes completem suas respectivas tarefas.
- Apresentar diferentes métodos para prevenir ou evitar deadlocks no sistema.

- Referência bibliográfica
  - Conceitos de sistemas operacionais com Java. 7ª edição. Silberchatz, Galvin and Gagne
  - Sistemas operacionais modernos. Andrew Tanenbaum.

Deadlocks: Agenda

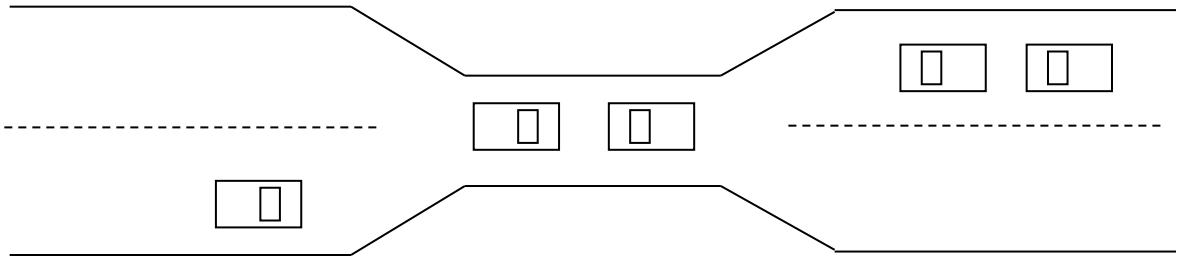
- O problema
- O modelo do sistema
- Caracterização de Deadlock
- Métodos para tratar Deadlocks
- Prevenção de Deadlock
- Evitar deadlock
- Detecção de Deadlock
- Recuperação de um Deadlock

O Problema do Deadlock

- Um conjunto de processos bloqueados, cada um retendo um recurso e esperando para adquirir um outro recurso retido por outro processo do conjunto.
- Examplo
  - O sistema tem 2 dois disco rigidos.
  - P_1 e P_2 retem, cada um, um disco rigido e necessita outro disco rigido.
- Examplo
  - Semaforos A e B, iniciados com 1

P_0	P_1
wait (A);	wait(B)
wait (B);	wait(A)

Exemplo da ponte estreita



- Tráfego na ponte somente numa única direção.
- Cada seção da ponte é um recurso.
- Um deadlock pode ser resolvido se um dos carros da a ré (liberar o recurso e voltar).
- Vários carros poderão ter que voltar, se um deadlock ocorrer.
- *Starvation* pode acontecer.

O Modelo do sistema

- Tipos de recursos $R_1, R_2, \ldots, R_m$

  Ciclos de CPU, espaço em memória, dispositivos I/O

- Cada tipo de recurso R_i tem W_i instâncias.
- Como um processo utiliza um recurso:
  - solicita
  - usa
  - libera

Caracterização do deadlock

Deadlock ocorre se quatro condições se dão ao mesmo tempo.

1. **Exclusão mútua:** somente um processo pode usar um recurso por vez.
2. **Reter e esperar:** um processo retém um recurso adquirido enquanto espera por outros que estão sendo usados por outro processo.
3. **Sem preempção:** um recurso só é liberado, pelo processo que o esteja usando, quando terminar a tarefa.
4. **Espera circular:** existe um conjunto de processos $\{P_0, P_1, \ldots, P_n\}$ esperando por um recurso, tal que P_0 espera por um recurso de P_1, P_1 espera por um recurso de P_2, …, P_{n-1} espera por um recurso de P_n, e P_n espera por um recurso de P_0.

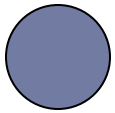
Grafo Alocacao de Recurso

Um conjunto de vértices V e um conjunto de arestas E.

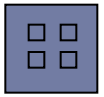
- V é particionado em dois tipos:
  - $P = \{P_1, P_2, \ldots, P_n\}$, o conjunto que consiste de todos os processos em execucao no sistema.
  - $R = \{R_1, R_2, \ldots, R_m\}$, o conjunto que consiste de todos os tipos de recursos do sistema.
- Solicitacao recurso: aresta direc. $P_1 \rightarrow R_j$
- Disponibilizacao recurso: aresta direc. $R_j \rightarrow P_i$

Grafo Alocacao de Recurso (Cont.)

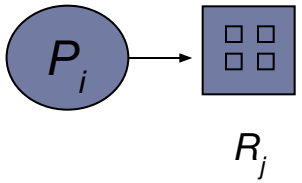
- Processo



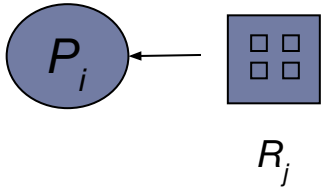
- Tipo de Recurso com 4 Instancias



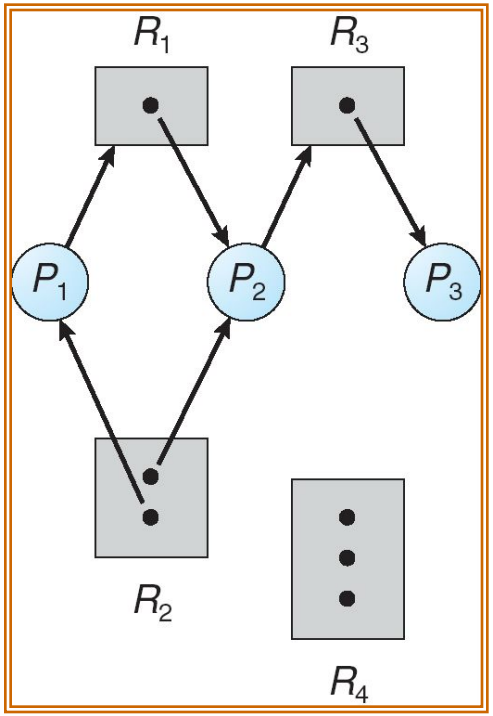
- P_i requisita instancia de R_j



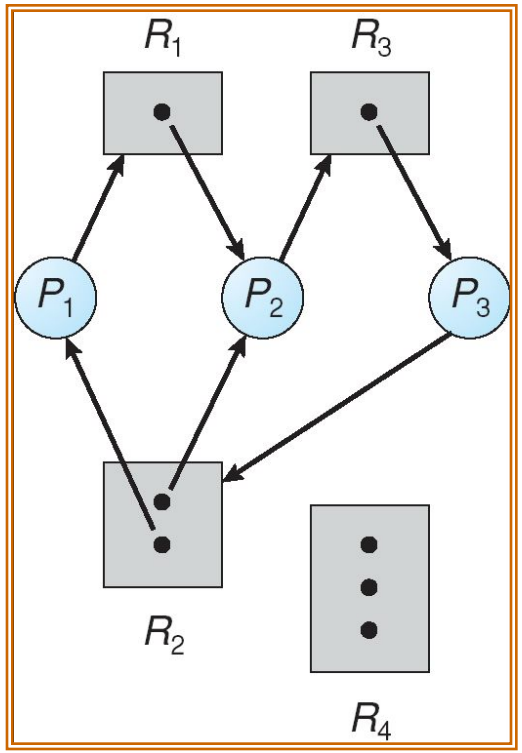
- P_i adquire um instancia de R_j



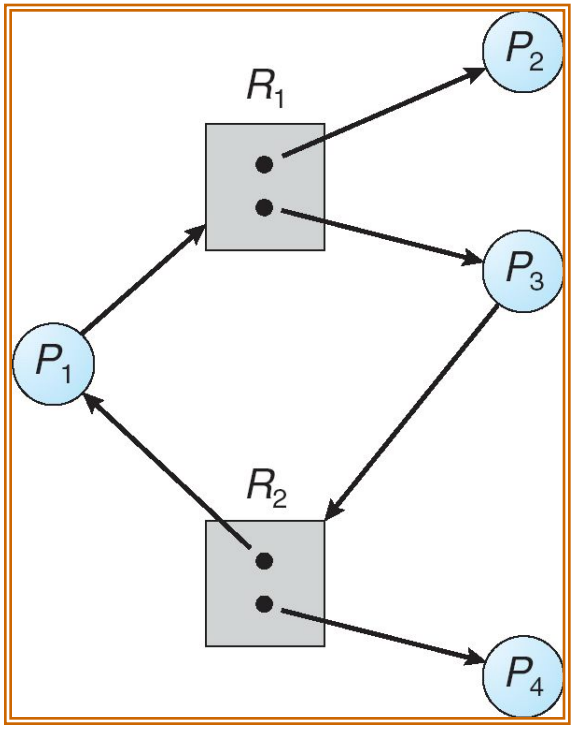
Exemplo de Grafo Alocação de Recurso



Grafo Alocacao de Recurso com Deadlock



Grafo Alocacao de Recurso com um ciclo porém sem Deadlock



Fatos Basicos

- Se o grafo não contém ciclos ⇒ não existe deadlock.
- Se o grafo contém ciclos ⇒
  - Se temos apenas uma instancia por tipo de recurso entao temos um deadlock.
  - Se temos varias instancias por tipo de recurso, entao POSSIVELMENTE temos um deadlock.

Exemplo de Deadlock em Java

```
class A implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         first.lock();
         // do something
            second.lock();
            // do something else
      }
      finally {
         first.unlock();
         second.unlock();
      }
   }
}
```

Thread A

```
class B implements Runnable
{
   private Lock first, second;

   public A(Lock first, Lock second) {
      this.first = first;
      this.second = second;
   }

   public void run() {
      try {
         second.lock();
         // do something
            first.lock();
            // do something else
      }
      finally {
         second.unlock();
         first.unlock();
      }
   }
}
```

Thread B

Exemplo de Deadlock em Java

```
public static void main(String arg[]) {
   Lock lockX = new ReentrantLock();
   Lock lockY = new ReentrantLock();

   Thread threadA = new Thread(new A(lockX,lockY));
   Thread threadB = new Thread(new B(lockX,lockY));

   threadA.start();
   threadB.start();
}
```

Deadlock é possivel se:

threadA -> lockY -> threadB -> lockX -> threadA

Tratando Deadlock em Java

```
public class ClockApplet extends Applet implements Runnable
{
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run() {
       while (true) {
          try {
             // sleep for 1 second
             Thread.sleep(1000);

             // repaint the date and time
             repaint();

             // see if we need to suspend ourself
             synchronized (mutex) {
                while (ok == false)
                   mutex.wait();
             }
          }
          catch (InterruptedException e) { }
       }
    }

    public void start() {
       // Figure 7.7
    }

    public void stop() {
       // Figure 7.7
    }

    public void paint(Graphics g) {
      g.drawString(new java.util.Date().toString(),10,30);
    }
}
```

Tratando Deadlock em Java

```
// this method is called when the applet is
// started or we return to the applet
public void start() {
    ok = true;

    if (clockThread == null) {
       clockThread = new Thread(this);
       clockThread.start();
    }
    else {
       synchronized(mutex) {
          mutex.notify();
       }
    }
}

// this method is called when we
// leave the page the applet is on
public void stop() {
    synchronized(mutex) {
       ok = false;
    }
}
```

Métodos para tratar deadlocks

- Garantir que o sistema *nunca* vai entrar num estado de deadlock.
- Permitir que ocorra um deadlock e então restaurar o sistema.
- Ignorar o problema e assumir que deadlocks nunca acontecem no sistema, usado pela maioria dos sistemas, incluindo UNIX.

Prevenção de Deadlock

Restringir as formas como uma solicitação pode ser feita.

- **Exclusão Mútua** – não necessária para recursos compartilháveis; mas valendo para recursos não compartilháveis.
- **Reter e esperar** – garantir que sempre que um processo solicite um recurso, não esteja usando outros.
  - Solicitação e alocação dos recursos necessários antes de iniciar a execução, ou somente quando não esteja usando algum recurso.
  - Conseqüências: Baixa utilização dos recursos; *starvation* pode acontecer.

Prevenção de Deadlock (Cont.)

- **Sem preempção** –
  - Se um processo que alocou alguns recursos, solicita outro e não pode ser atendido, então todos os recursos desse processo são liberados
  - Processo será reiniciado somente quando ele possa reaver todos os recursos, incluindo o novo recurso solicitado.
- **Espera Circular** – ordenar totalmente todos os tipos de recursos, e impor que cada solicitação de recurso seja feita em ordem.

Evitar Deadlock

O Sistema deve contar com informações *a priori*

- Simples: cada processo declara o número máximo de cada tipo de recurso necessário.

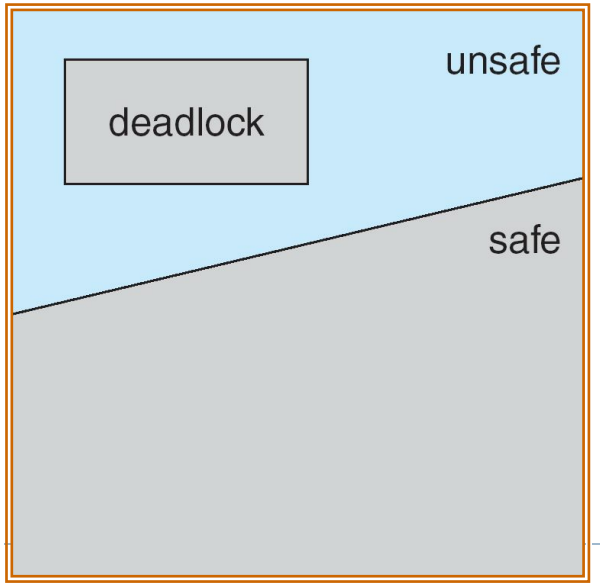
- Dinâmico: algoritmo verifica o **estado de alocação de recursos** para garantir que não ocorra uma espera circular.
  - Estado de alocação de recursos: número de recursos alocados e disponíveis, máximo demandas dos processos.

Estado seguro

- O sistema está num estado seguro se existe uma seqüência de todos os processos $<P_1, P_2, \ldots, P_n>$ tal que os recursos requeridos por cada P_i estão disponíveis ou estão em uso pelo processo P_j, sendo $j < i$.
- Ou seja:
  - Se os recursos que P_i necessita não estão disponíveis, ele espera até terminar P_j
  - Quando P_j terminar, P_i pode obter os recursos, iniciar a execução e concluir.
  - Quando P_i termina, P_{i+1} pode obter os recursos e assim sucessivamente.

Fatos básicos

- Sistema em estado seguro ⇒ sem deadlocks.
- Sistema em estado inseguro ⇒ possibilidade de deadlock
- Evitar ⇒ garantir que o sistema não entre em estado inseguro.



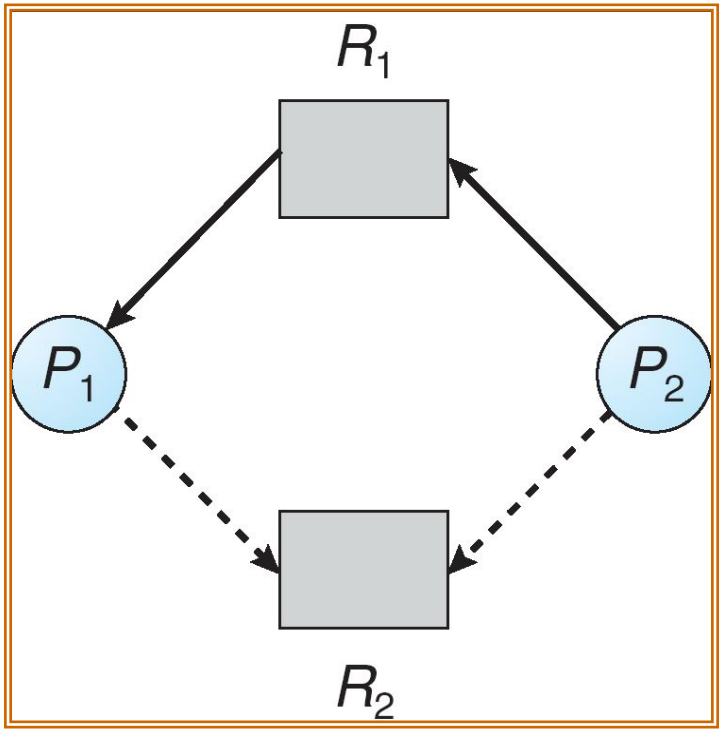
Evitar Deadlock - Algoritmos

- Uma instância de cada recurso:
  - Grafo de alocação de recursos

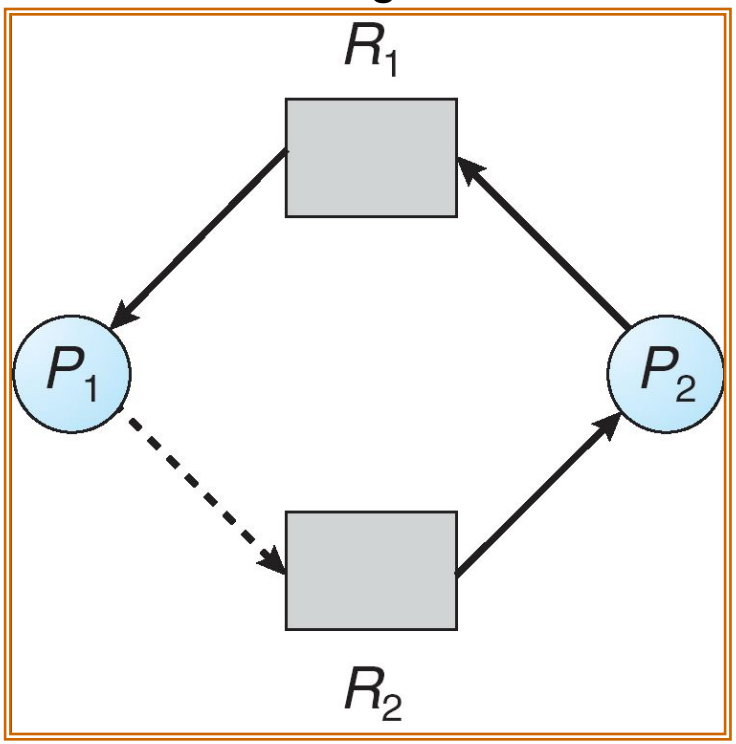
- Várias instâncias de um recurso:
  - Algoritmo do banqueiro (Edsger Dijkstra)

Evitar deadlock –

Grafo de alocação de recursos



Estado inseguro



Evitar deadlock –

Algoritmo do banqueiro

- Instâncias múltiplas.
- Cada processo deve requerer *a priori* os recursos.
- Quando um processo solicita um recurso, talvez tenha que esperar.
- Após obter todos os recursos que precisa, um processo deve devolvê-los em um tempo finito.

Evitar Deadlock –

Algoritmo do banqueiro - Estruturas de dados

n = número de processos, *m* =número de tipo de recursos, R tipo de recurso.

- ***Available:*** Vetor de tamanho *m*.
  - Se available [*j*] = *k*, existem *k* instâncias disponíveis de R_j.
- ***Max:*** matriz *n* x *m*.
  - Se *Max* [*i,j*] = *k*, entao processo P_i poderá solicitar até *k* instâncias de R_j.
- ***Allocation:*** matriz *n* x *m*.
  - Se Allocation[*i,j*] = *k*, P_i alocou *k* instâncias de R_j.
- ***Need:*** matriz *n* x *m*.
  - Se *Need*[*i,j*] = *k*, P_i precisa mais *k* instâncias de R_j para terminar.

Need [*i,j*] = *Max*[*i,j*] – *Allocation* [*i,j*].

Algoritmo de Segurança

1. ***Work*** e ***Finish*** são vetores de tamanho *m* recursos e *n* *processos*, respectivamente. Inicializar:
   Work = *Available*
   Finish [*i*] = *false* for *i* = 0, 1, …, *n*- 1.
2. Encontrar *i* tal que:
   (a) *Finish* [*i*] = *false*
   (b) $Need_i \leq Work$
   IF *i* não existe, vá ao passo 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   Ir ao passo 2.
4. IF *Finish* [*i*] == true para todo *i*, estado é seguro.

Exemplo do algoritmo do banqueiro

- 5 processos: P_0 a P_4;

  3 Tipos de recursos:

  A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Ordem Execução: P_1

Exemplo do algoritmo do banqueiro

- 5 processos: P_0 a P_4;

  3 Tipos de recursos:

  A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	5 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Ordem Execução: $P_1 > P_3$

Exemplo do algoritmo do banqueiro

- 5 processos: P_0 a P_4;

  3 Tipos de recursos:

  A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	7 4 3
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Ordem Execução: $P_1 > P_3 > P_0$

Exemplo do algoritmo do banqueiro

- 5 processos: P_0 a P_4;

  3 Tipos de recursos:

  A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	7 5 3
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Ordem Execução: $P_1 > P_3 > P_0 > P_2$

Exemplo do algoritmo do banqueiro

- 5 processos: P_0 a P_4;

  3 Tipos de recursos:

  A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	10 5 5
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Ordem Execução: $P_1 > P_3 > P_0 > P_2 > P_4$

Exemplo do algoritmo do banqueiro

- 5 processos: P_0 a P_4;

  3 Tipos de recursos:

  A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	10 5 7
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Ordem Execução: $P_1 > P_3 > P_0 > P_2 > P_4 >$ *SEGURO !!*

Evitar deadlock –

Algoritmo solicitação recurso para processo P_i

Request = vetor solicitações processo P_i.
IF $Request_i[j] = k$ THEN processo P_i pede k instâncias do tipo R_j.

1. IF $Request_i \leq Need_i$ THEN ir passo 2
   ELSE Erro, processo excedeu limite máximo de requisições.
2. IF $Request_i \leq Available$, THEN ir passo 3
   ELSE P_i espera por recursos
3. Simular alocação de recursos para P_i modificando o estado:

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

- *IF seguro⇒ recursos são alocados a P_i*
- *IF inseguro ⇒ P_i espera, estado de solicitacao-recurso volta ao estado anterior*

Evitar deadlock -

Exemplo banqueiro: P_1 solicita (1,0,2)

- Verifique que Request ≤ Available (isto é, (1,0,2) ≤ (1,2,2) ⇒ true.

ANTES

A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

T_0:	*Allocation*	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

DEPOIS

	Allocation	*Need*	*Work*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Evitar deadlock -

Exemplo banqueiro: P_1 solicita (1,0,2)

- Resultado execução do algoritmo de segurança: < P_1, P_3, P_4, P_0, P_2> satisfaz requerimentos.
- Desvantagens
  - Conhecer a priori todas as necessidades
  - Número estático de processos
  - Esperar que um processo termine pode não ser aceitável na "vida real".

Evitar deadlock -

Exemplo banqueiro: P_1 solicita (1,0,2)

- Verifique que Request ≤ Available (isto é, (1,0,2) ≤ (3,3,2) ⇒ true.

	Allocation	*Need*	*Work*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Resultado execução do algoritmo de segurança: < P_1, P_3, P_4, P_0, P_2> satisfaz requerimentos.
- Desvantagens
  - Conhecer a priori todas as necessidades
  - Número estático de processos
  - Esperar que um processo termine pode não ser aceitável na "vida real".

Exercício

- Dado o exemplo da execução do algoritmo do banqueiro para evitar deadlocks, apresentada nesta aula, verifique se o sistema estará em estado seguro após as seguintes solicitações dos processos P4 e P0:

  (a) P_4 realiza a seguinte solicitação (3,3,0)

  (b) P_0 realiza a seguinte solicitação (0,2,0)

A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

T_0:	*Allocation*	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	3 3 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

Resposta Exercício

(a) P_4 realiza a seguinte solicitação (3,3,0)

A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 1 0	7 5 3	7 4 3	0 0 2
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	3 3 2	4 3 3	1 0 1	

ESTADO INSEGURO - NINGUEM CONSEGUE EXECUTAR

Resposta exercício

(b) P_0 realiza a seguinte solicitação (0,2,0)

A (10 instâncias), *B* (5 instâncias), *C* (7 instâncias).

- T_0:

	Allocation	*Max*	*Need*	*Work*
	A B C	*A B C*	*A B C*	*A B C*
P_0	0 3 0	7 5 3	7 2 3	3 1 2, 5 2 3, 7 2 3, 7 5 3, 10 5 5,
P_1	2 0 0	3 2 2	1 2 2	
P_2	3 0 2	9 0 2	6 0 0	
P_3	2 1 1	2 2 2	0 1 1	
P_4	0 0 2	4 3 3	4 3 1	

SEQ: P3- P1- P0 - P2 - P4 - ESTÁ SEGURO

Exercicio

- Suponha que um sistema esteja em estado inseguro. Mostre que é possivel para os processos completarem sua execução sem entrar em um estado de deadlock.

Exercicio - Solução

- Verifique a seguinte situaçao onde o sistema possui 12 recursos

	Max	Current	Need
P0	10	5	5
P1	4	2	2
P2	9	3	6

- Nao podemos garantir que os processos P0 e P2 podem completar, mas é possivel que um processo possa liberar recursos antes de requesitar outro. Por exemplo, o processo P2 pode liberar 1 recurso, aumentando o numero de recursos para 5

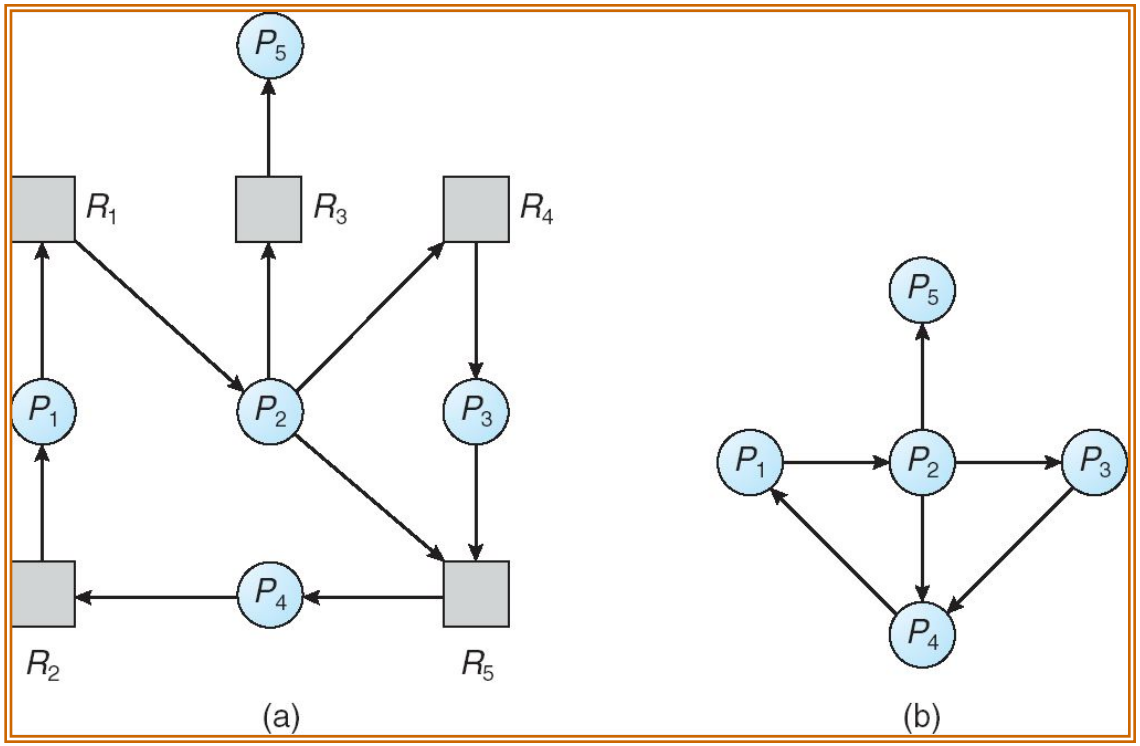
Detecção de Deadlock

- Permite sistema entrar em deadlock
- Algoritmo de Detecção
- Sistema de Recuperação

Uma unica instancia de cada tipo de recurso

- Manter o grafo de *Espera/Por*
  - Nós são processos.
  - $P_i \rightarrow P_j$ se P_i *esta aguardando por* P_j.
- Periodicamente invoca um algoritmo que procura por um ciclo no grafo. Se existe um ciclo, então existe um deadlock.
- Um algoritmo que detecta um ciclo no grafo requer uma ordem de n^2 operações onde n é o número de vertices no grafo.

Grafo de Alocação-Recurso e Grafo de Espera-Por



Grafo de Alocação-Recurso

Corresponden grafo de Espera-Por

Diversas Instancias de um tipo de recurso

- ***Available**:* Um vetor de tamanho *m* indica o numero de recursos disponiveis de cada tipo.
- ***Allocation**:* Uma matrix *n* x *m* define o numero de recursos de cada tipo alocado para cada processo.
- ***Request**:* Uma matrix *n* x *m* indica a requisição corrente de cada processo. Se *Request* [i_j] = *k*, então cada processo P_i esta requisitando *k* mais instancias do tipo de recurso R_j.

Algoritmo de Detecção

1. Seja *Work* e *Finish* vetores de tamanho *m* and *n*, respectivamente, Inicie-os com:
   (a) *Work* = *Available*
   (b) For *i* = 1,2, …, *n*, se $Allocation_i \neq 0$, então *Finish*[i] = false; otherwise, *Finish*[i] = *true*.
2. Encontre um indice *i* tal que:
   (a) *Finish*[*i*] == *false*
   (b) $Request_i \leq Work$
   Se nenhum *i* existe, go to step 4.
3. $Work = Work + Allocation_i$
   Finish[*i*] = *true*
   go to step 2.
4. If *Finish*[*i*] == false, para algum *i*, $1 \leq i \leq n$, entao o sistema esta' em estado de deadlock. Ainda mais, se *Finish*[*i*] == *false*, entao P_i esta em deadlock.

Este algoritmo requer uma ordem de $O(m \times n^2)$ operações para detectar se o sistema esta em estado de deadlock.

Exemplo

- 5 processos P_0 até P_4; 3 tipos de recursos
  A (7 instancias), *B* (2 instancias), and *C* (6 instancias).
- No tempo T_0:

	Allocation	*Request*	*Work*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 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	

- Sequencia <P_0, P_2, P_3, P_1, P_4> resultara' em *Finish*[*i*] = true para todo *i*.

Exemplo (Cont.)

- P_2 requisita um instancia do tipo *C*.

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 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	

- Estado do Sistema ?
  - Pode solicitar recursos retidos pelo processo P_0, mas os recursos são insuficientes para atender as requisicoes dos outros processos.
  - Deadlock existe, consistindo dos processsos P_1, P_2, P_3, e P_4.

Uso do Algoritmo de Detecção

- Quando, e com que frequência invocar, depende de:
  - Com que frequência um deadlock é provável de acontecer?
  - Quantos processos serão desfeitos?
- Se o algoritmo de detecção é invocado arbitrariamente, existem vários ciclos no grafo de recursos e então não é possível dizer quais dos processos causou deadlock.

Recuperando de um Deadlock: Finalização de um Processo

- Abortar todos os processos em deadlock.
- Abortar um processo a cada momento até que o ciclo de deadlock seja eliminado.
- Em qual ordem nos devemos abortar?
  - Prioridade do processos.
  - Quanto tempo o processo gastou, e quanto ainda falta para este processo terminar.
  - Recursos que o processo usou.
  - Recursos que o processo precisa para terminar.
  - Quantos processos serao necessarios para serem finalizados.
  - O processo é interativo ou batch?

Recuperando de um Deadlock: Preempção de Recurso

- Selecionar uma vitima – minimizar custo.
- Desfazer (Rollback) – retornar para algum estado seguro, reiniciar o processo para este estado.
- Starvation – algum processo pode ser sempre selecionado como vítima.