Sistemas Operacionais

Deadlock

Deadlocks

- Recursos: hardware ou informação
- Preemptivo X não preemptivo

- Uso do Recurso:
  - Pedido (Request ou Open)
  - Uso
  - Liberação

- “Um conjunto de processos está em *deadlock* se cada processo no conjunto está esperando por um *evento* que somente um outro processo do conjunto pode causar.”

Deadlocks

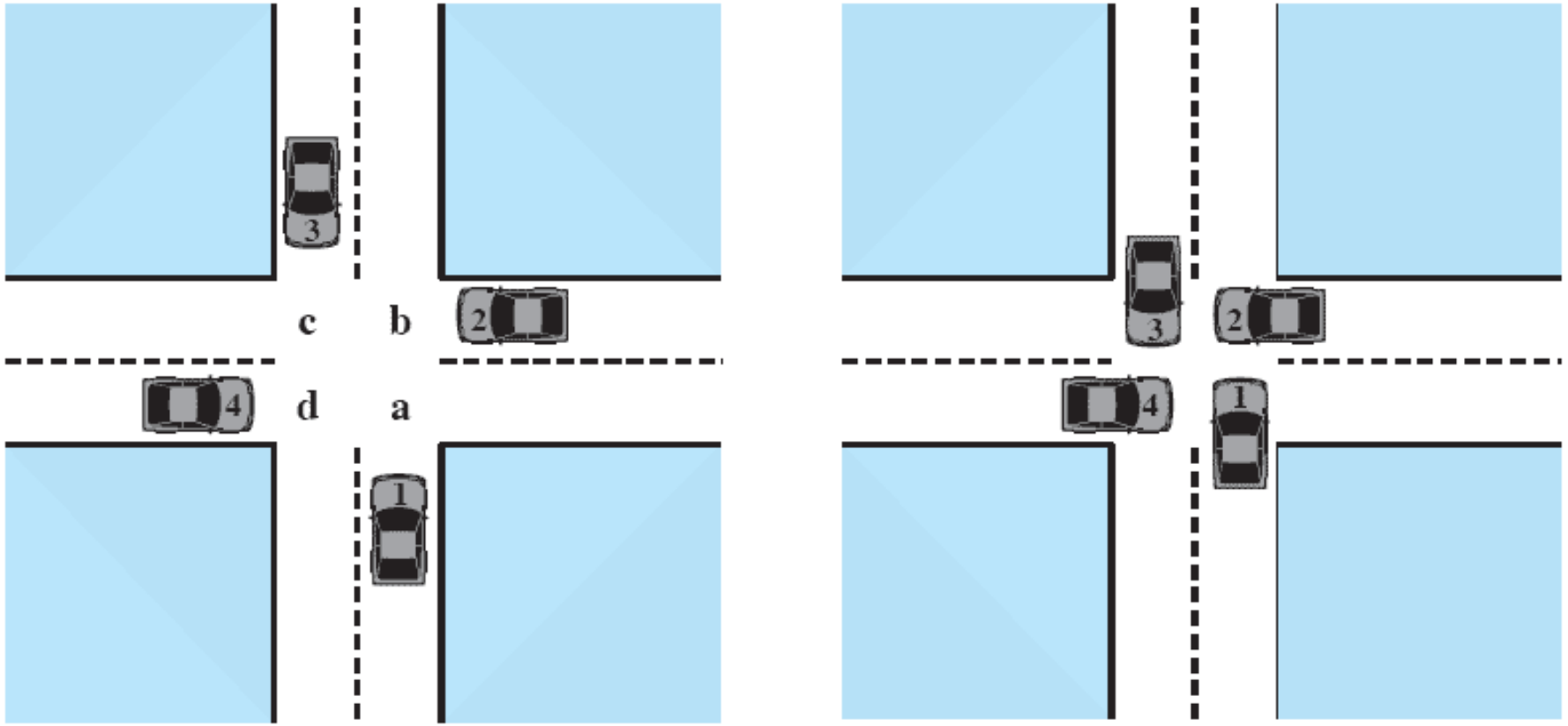


Figure 6.1 Illustration of Deadlock

Deadlocks

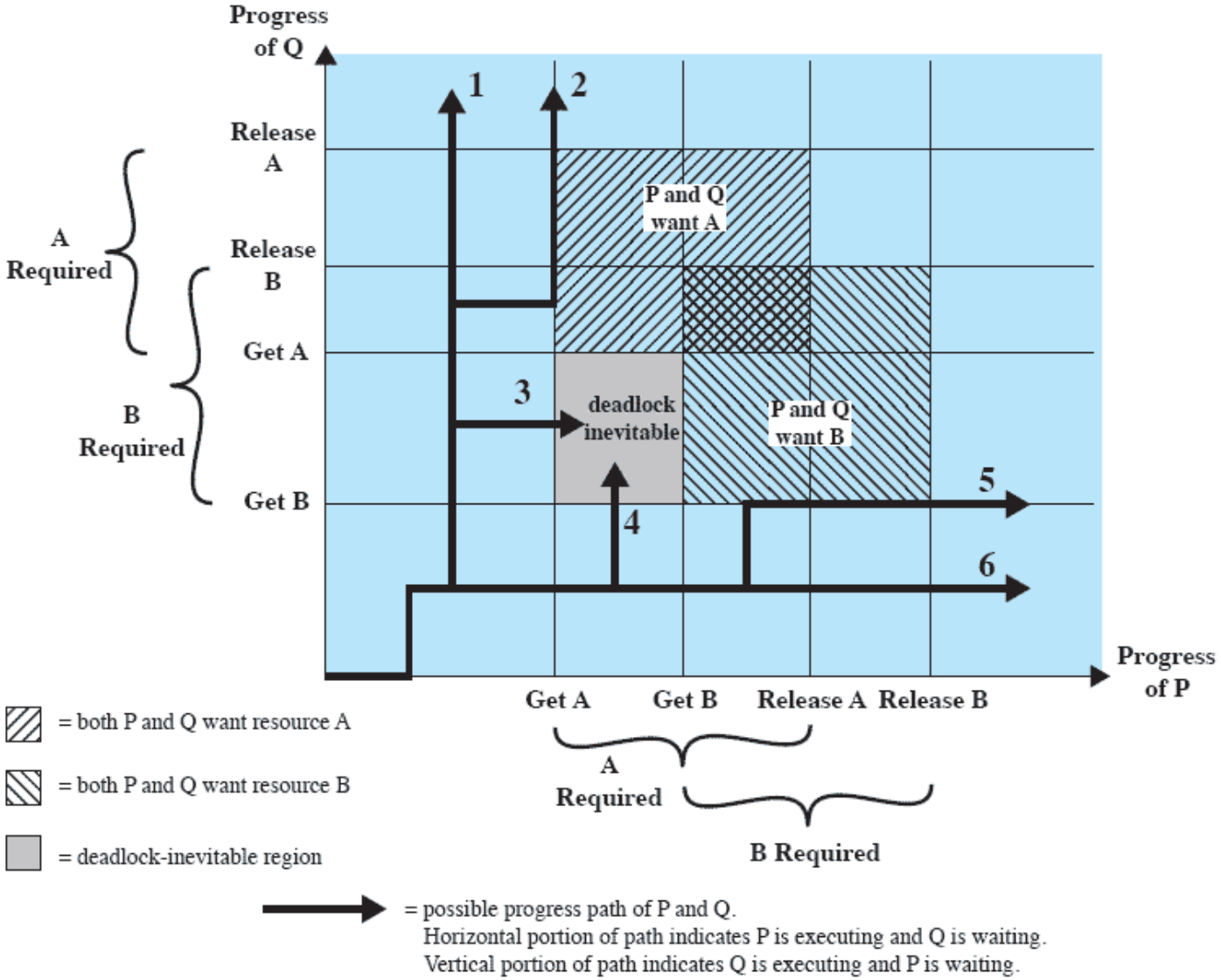


Figure 6.2 Example of Deadlock

Deadlocks

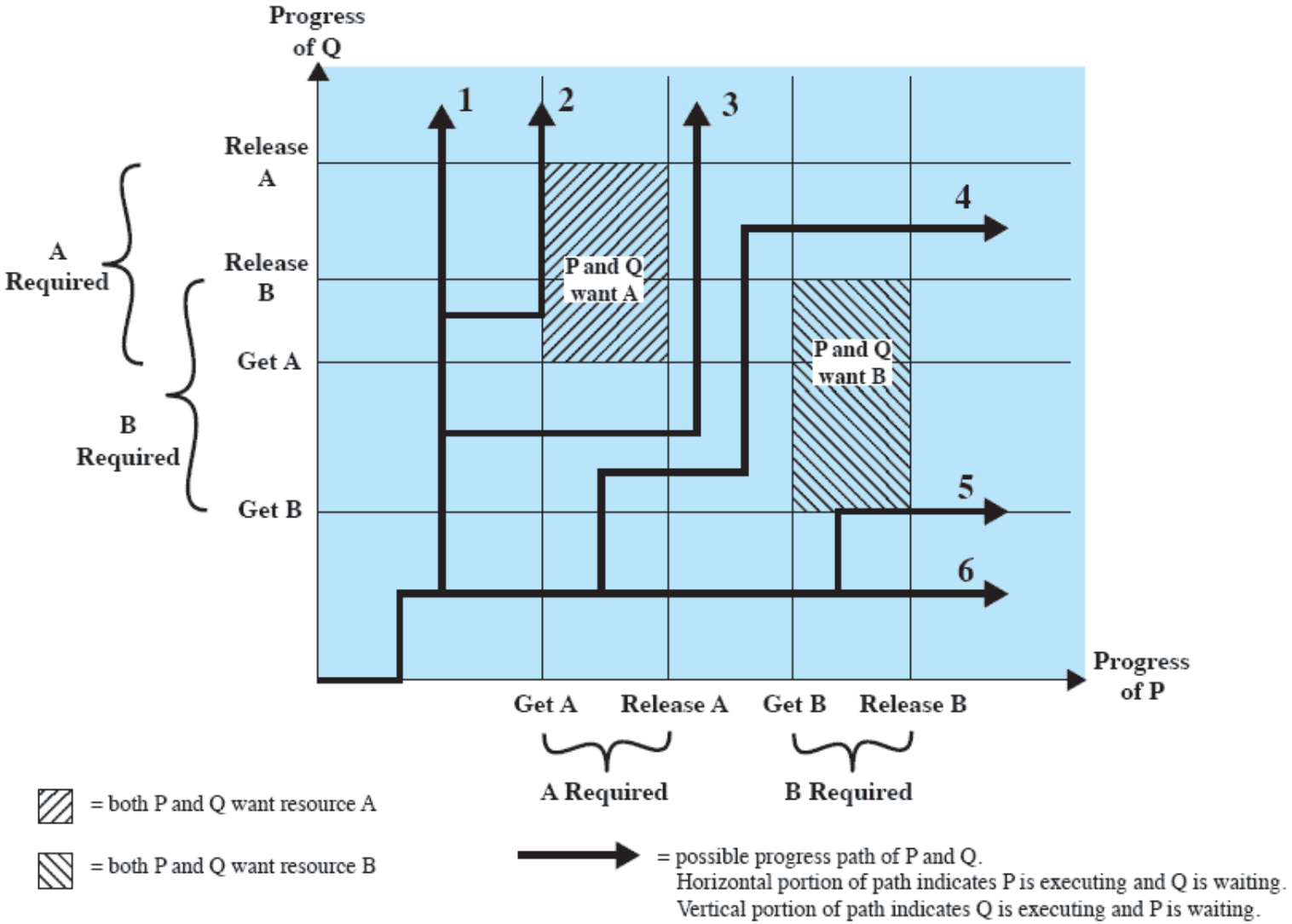


Figure 6.3 Example of No Deadlock [BACO03]

Deadlock – recursos reutilizáveis

Process P

Step	Action
p_0	Request (D)
p_1	Lock (D)
p_2	Request (T)
p_3	Lock (T)
p_4	Perform function
p_5	Unlock (D)
p_6	Unlock (T)

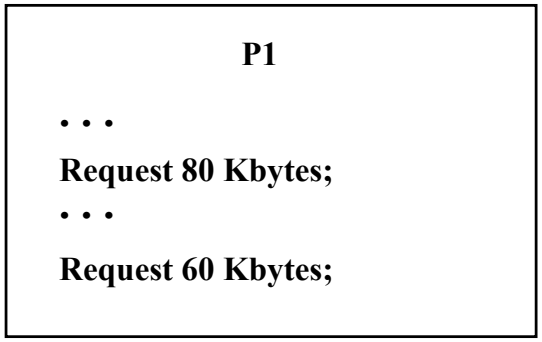
Process Q

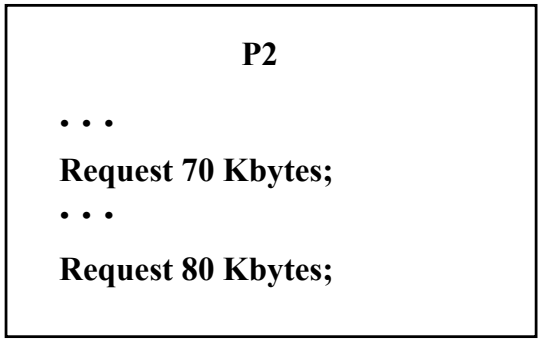
Step	Action
q_0	Request (T)
q_1	Lock (T)
q_2	Request (D)
q_3	Lock (D)
q_4	Perform function
q_5	Unlock (T)
q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

Deadlock – recursos reutilizáveis

- 200kBytes de memória estão disponíveis para alocação, mas a seguinte sequencia de pedidos é realizada





- se o segundo pedido é realizado sem liberação, pode causar deadlock
  - se for um pedido bloqueante

Deadlocks

- As seguintes condições podem levar a ocorrência de deadlock
  - Exclusão Mútua
    - Somente um processo usa um recurso de cada vez
  - Obtenção e Espera
    - Um processo aloca recursos enquanto espera por outros recursos
  - Não preempção dos recursos
    - Recursos não sofrem preempção – não são tirados de processos aos quais já foram alocados

Deadlocks

- As condições acima são necessárias mas não suficientes
  - Cadeia fechada recurso+processo

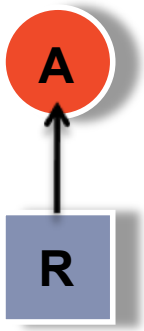
Deadlocks

Possibilidade de Deadlock	Existência de Deadlock
Exclusão mútua	Exclusão mútua
Sem preempção de recursos	Sem preempção de recursos
Obtenção e espera	Obtenção e espera
	Cadeia fechada

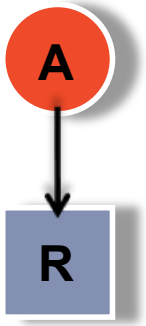
Deadlocks

Modelo - Grafo

- Processos são círculos e recursos são quadrados.
- O recurso R está alocado ao processo A:
  - arco de R para A



  - O processo A pede o recurso R

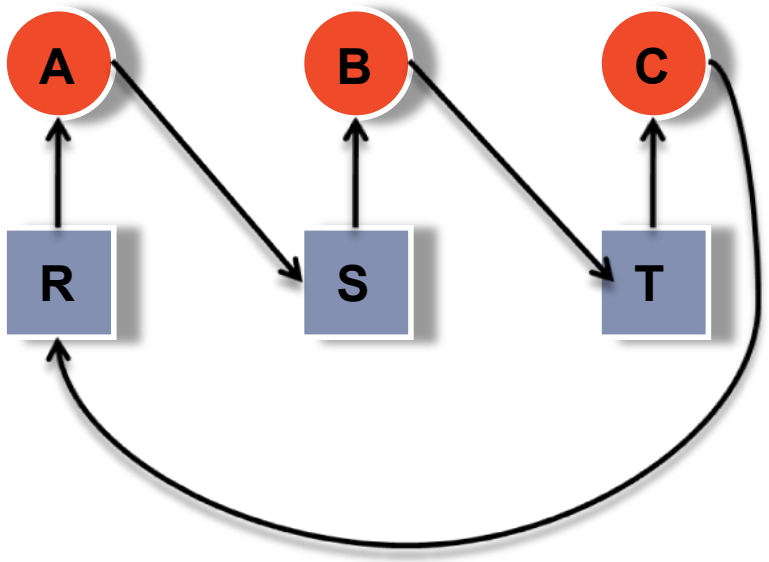


Deadlocks

Uma cadeia circular – não é um exemplo simples

1. A pede R
2. B pede S
3. C pede T
4. A pede S
5. B pede T
6. C pede R

deadlock



SO – como gerenciar a deadlocks

- SO impor uma ordem na alocação de recursos para prevenção?
- Como saber a alocação antecipada?
- Montar um grafo de recursos+ processos?
- Que mecanismo especificar?

SO – como gerenciar a deadlocks

- prevenção de *deadlocks*
- evitar *deadlock*
- detecção de *deadlock*

SO – como gerenciar a deadlocks

- Prevenção de *deadlocks*
  - através de uma política que evita uma das quatro condições
  - pode ser conservativo
    - Não prover recursos para evitar *deadlocks*
    - Isso pode tornar o sistema não eficiente

SO – como gerenciar a deadlocks

- mecanismo executado dinamicamente: avaliação dinâmica das possíveis alocações dos recursos pelos processos. Duas alternativas são possíveis:
  - Não começar um processo se levar a *deadlock*
  - Não garantir a alocação de recurso se leva a *deadlock*

Evitando Deadlocks

- Evitar dinamicamente: as três condições podem acontecer, mas a quarta é evitada através de decisões dinâmicas. Duas alternativas são possíveis:
  - Não começar um processo se levar a *deadlock*
  - Não garantir a alocação de recurso se leva a *deadlock*
  - exclusão mútua a nível de SO
  - não garantir preempção de recursos

Evitando Deadlocks

- um exemplo, através de *banker's algorithm*
- O estado do sistema é a alocação atual de recursos
- *estado* seguro
  - existe uma sequencia de alocações que não leva ao deadlock
- *estado inseguro*
  - caso contrário

Evitando Deadlocks

Estrutura de dados

- claim (recursos X processos)
  - o requisito máximo de recursos de cada processo
  - para evitar deadlock deve ser previamente definido
- allocation (recursos X processos)
  - alocação atual de cada recurso
- available (recursos)
  - disponibilidade de recursos

Evitando Deadlocks

Na verdade estado do sistema pode ser representado

- claim (recursos X processos)
- allocation (recursos X processos)
- available (recursos)

Estado Seguro

- R = total de instâncias de cada recursos
- V = o que ficou disponível depois da alocação em A
- C-A = o que ainda será alocado durante a execução de cada processo

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Estado Seguro

- P2 completa sua execução
- recursos de P2 se tornam disponíveis outra vez
- vai haver problema de alocação para os processos restantes?
- e se os recursos de P1 forem alocados?
  - ainda precisa de (2,2,2), que está disponível. Pode alocar!.

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Estado Seguro

- Quando P1 terminar, retorna os recursos para disponibilidade
- P3 pode executar suas alocações?
  - pela disponibilidade pode! Ao terminar, libera!

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

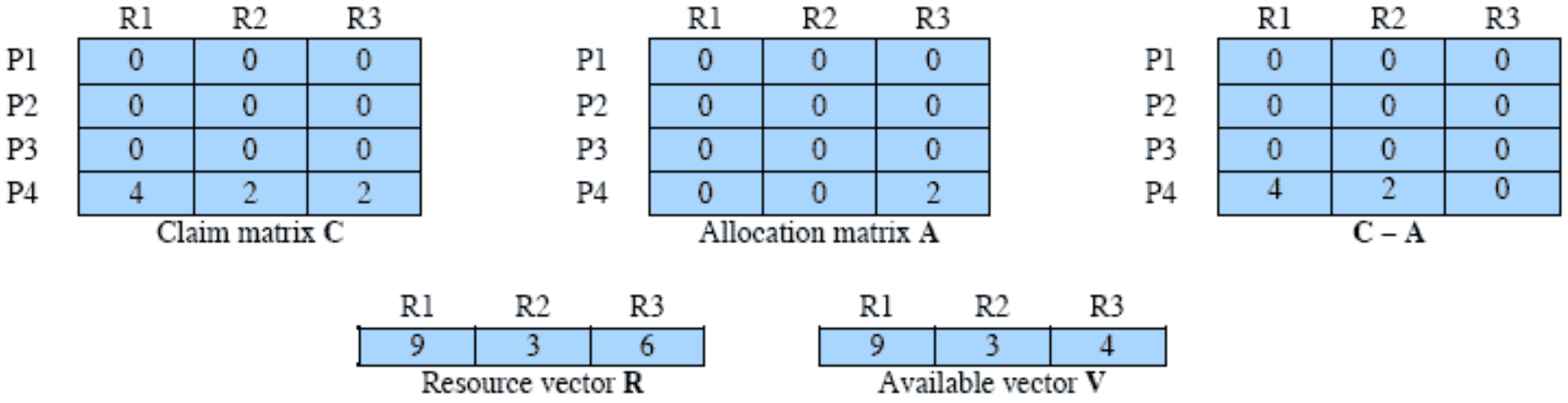
R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Estado Seguro

- Quando P3 terminar
- P4 pode alocar e terminar com sucesso
- Logo, o estado inicial é seguro!



(d) P3 runs to completion

Estado Seguro

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

- se o pedido de P1 for garantido, levará a deadlock, pois agora, outros processos precisam de 1 unidade de R1

Evitando Deadlocks

```
struct state {
        int resource[m];
        int available[m];
        int claim[n][m];
        int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                                  /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                                  /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

Evitando Deadlocks

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                          /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

Figure 6.9 Deadlock Avoidance Logic

Detecção de Deadlocks

- Recursos são garantidos sempre que possível, mas os pedidos de alocação não são recusados como no caso de prevenção de deadlock (que é muito conservativo)
  - O SO pode checar a possibilidade a cada pedido de alocação de recurso (ou de forma menos frequente)
  - baseado em informações incrementais do sistema
    - tende a ser mais eficiente
  - medidas são também tomadas dinamicamente

Detecção de Deadlocks

Informações necessárias:

- Matriz de alocação A
  - $A_{i,j}$ - alocação corrente em unidades do recurso *j* ao processo *i*
- *Avail* = $(V_1, V_2, ..., V_m)$: disponibilidade de instâncias de cada recurso V_i não alocada a nenhum processo
- Q: matriz de pedidos
  - $Q_{i,j}$ – é a quantidade de recursos do tipo *j* pedida pelo processo *i*

Algoritmo de detecção de *deadlock*

- Seja *A* uma matriz de alocação
  - onde $A_{i,j}$ é a alocação em unidades do recurso *j* ao processo *i*

	R_1	R_2	R_3	R_4	R_5
P_1	1	2	0	1	0
P_2	1	0	1	1	0

- Matriz de pedidos *Q*
  - $Q_{i,j}$ – é a quantidade de recursos do tipo *j* pedida pelo processo *i*
- *Avail* = (V_1, V_2, ..., V_m): quantidade de cada recurso não alocada a nenhum processo (disponibilidade do recurso)

Algoritmo de detecção de *deadlock*

- Inicialmente, todos os processos são considerados *unmarked*
- Idéia principal: satisfazer pedidos de processos de acordo com a disponibilidade
  - garantir alocação até o final da execução dos respectivos processos
- Se ainda existirem processos *unmarked* ao final do algoritmo
  - estes estão em *deadlock* - a detecção
  - esse algoritmo não previne deadlock – algo tem que ser feito

Algoritmo de detecção de *deadlock*

① Marque cada processo que tenha uma linha em *A* somente com zeros

- nada foi alocado para aquele processo

② Inicializar *W* (temporário) com o vetor de disponibilidade *Avail*

③ Identifique um índice *i* tal que o processo *i* não esteja marcado é a *i*-ésima linha de *Q* é menor ou igual a *W*

$$Q_{i,j} \leq W_k \qquad 1 \leq k \leq m$$

Se tal linha não existir, termine (nada pode ser atendido)

④ Se tal linha existe, marque o processo *i* e adicione a linha correspondente da matriz de alocação *A* ao vetor *W*

$$W_k = W_k + A_{i,j}$$

Se tal linha não existir, volte para o passo 3 para o próxima pedido Q e atualizando *Avail*

Algoritmo de detecção de *deadlock*

- Significado – os recursos são alocados sem causar deadlock e depois são liberados (para a próxima iteração??)
  - Ao identificar um processo em (3), os recursos que estão disponíveis são alocados
  - Ao somar os já alocados em *A* com os disponíveis em *W* e considerá-los disponíveis
    - Processo terminou e pode liberar os recursos, sem causar *deadlock*

Algoritmo de detecção de *deadlock*

- *Deadlock* está ocorrendo se e somente se
  - existem processos não marcados ao final do algoritmo
  - Cada processo não marcado está em *deadlock*
- O algoritmo se baseia no fato de que:
  - Os recursos requisitados serão garantidos para o processo
  - O processo executará até o final com os recursos necessários alocados
- O algoritmo não previne a ocorrência de *deadlock*

Algoritmo de detecção de *deadlock*

① Marque cada processo que tenha uma linha em *A* somente com zeros

- nada foi alocado para aquele processo

② Inicializar *W* (temporário) com o vetor de disponibilidade *Avail*

③ Identifique um índice *i* tal que o processo *i* não esteja marcado é a *i*-ésima linha de *Q* é menor ou igual a *W*

$$Q_{i,j} \leq W_k \qquad 1 \leq k \leq m$$

Se tal linha não existir, termine (nada pode ser atendido)

④ Se tal linha existe, marque o processo *i* e adicione a linha correspondente da matriz de alocação *A* ao vetor *W*

$$W_k = W_k + A_{i,j}$$

Se tal linha não existir, volte para o passo 3

Algoritmo de detecção de *deadlock*

matriz de alocação A

	R_1	R_2	R_3	R_4	R_5	
P_1	1	0	1	1	0	não marcado
P_2	1	1	0	0	0	não marcado
P_3	0	0	0	1	0	não marcado
P_4	0	0	0	0	0	**Marcado**

vetor *Avail* - o vetor *W* recebe esses valores

R_1	R_2	R_3	R_4	R_5
0	0	0	0	1

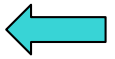
Passo 1

Passo 2

matriz *Q* de pedidos

	R_1	R_2	R_3	R_4	R_5
P_1	0	1	0	0	1
P_2	0	0	1	0	1
P_3	0	0	0	0	1
P_4	1	0	1	0	1

Passo 3: $Q_{i,j} \leq W_k$



Algoritmo de detecção de *deadlock*

$W = W + A_{ij}$

Passo 4

	R_1	R_2	R_3	R_4	R_5
W	0	0	0	0	1
$A_{3,j}$	0	0	0	1	0
W	0	0	0	1	1

e o processo P_3 se torna marcado

- *W* resultante = P_3 já tinha recursos alocados, conseguiu alocar mais, e ao terminar libera recursos.
- Próxima iteração: mais processos serão marcados?

- Processos P_1 e P_2 estão em deadlock (não são atendidos)

Abordagens de Recuperação após Detecção de processos em *Deadlock*

- Abortar os processos em *deadlock* – muito comum
- Recomeçar todos os processos em *deadlock* com algum *checkpointing* salvo anteriormente
  - *Checkpointing:* imagem da memória do processo
  - Mecanismos de *rollback* e *restart* necessários
  - Perigo de dar *deadlock* outra vez
- Recomeçar processo a processo em *deadlock*, executando o algoritmo de detecção a cada processo escolhido

Abordagens de Recuperação após Detecção de processos em *Deadlock*

- Preempção de recurso a recurso, executando o algoritmo de detecção a cada preempção
  - Processos que perderam recursos podem ter que retornar a estado anterior ao de alocação do recurso

Abordagens de Recuperação após Detecção de processos em *Deadlock*

- Qual processo em *deadlock* escolher?
  - Aquele que menos consumiu processador
  - Produziu menos resultados
  - Menos recursos alocados
  - Menor prioridade

Estratégias Integradas - *deadlock*

- Vantagens e desvantagens em previsão e detecção. Então, o que fazer?
  - Depende da situação

Pode-se

- Agrupar recursos em classes
- Usar uma estratégia para prevenir a ocorrência de uma cadeia circular entre as classes
- Dentro de cada classe, utilizar um outro algoritmo que está mais de acordo com aquela classe de recursos

Estratégias Integradas - *deadlock*

- Exemplo:
  a) Blocos de memória no dispositivo secundário para ser utilizado em swap
  b) Recursos de processos (como dispositivos)
  c) Memória principal
  d) Recursos internos: canais de I/O
- A ordem acima representa a ordem de alocação das classes (que está de acordo com o ciclo de vida de um processo, em geral)

Estratégias Integradas - *deadlock*

- Exemplo – dentro de cada classe

  a) Alocar tudo que for necessário de uma vez

  b) Evitar deadlock (não tem o recurso, não pode executar no momento)

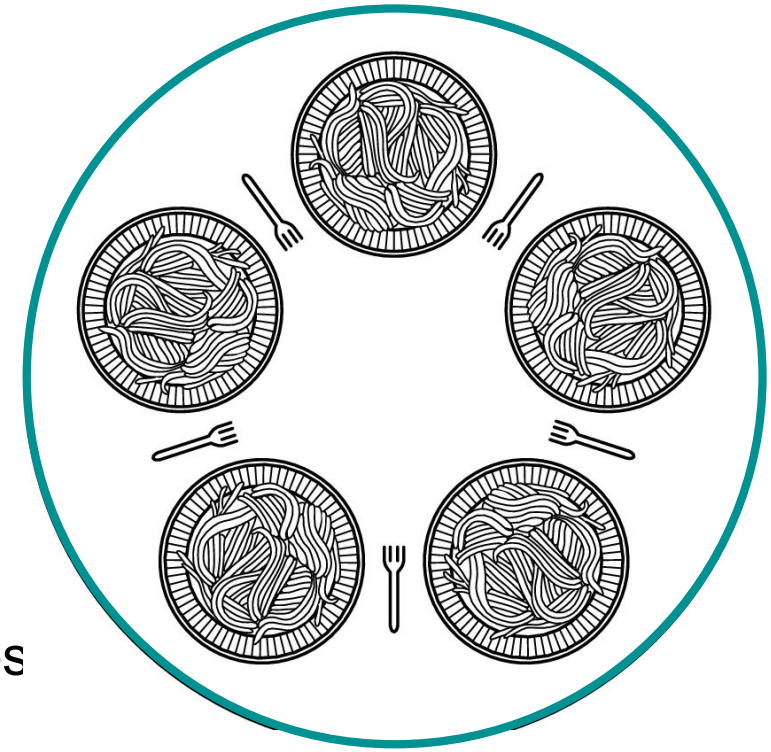
  c) Pode-se fazer swap para evitar deadlock que venha a ser previsto

  d) Ordenar os pedidos de recursos podem prevenir da ocorrência de deadlock

Outro Problema Clássico de Sincronização

- Jantar dos Filósofos – representa solução para dois tipos de problemas que ocorre na sua definição
  - deadlock
  - starvation

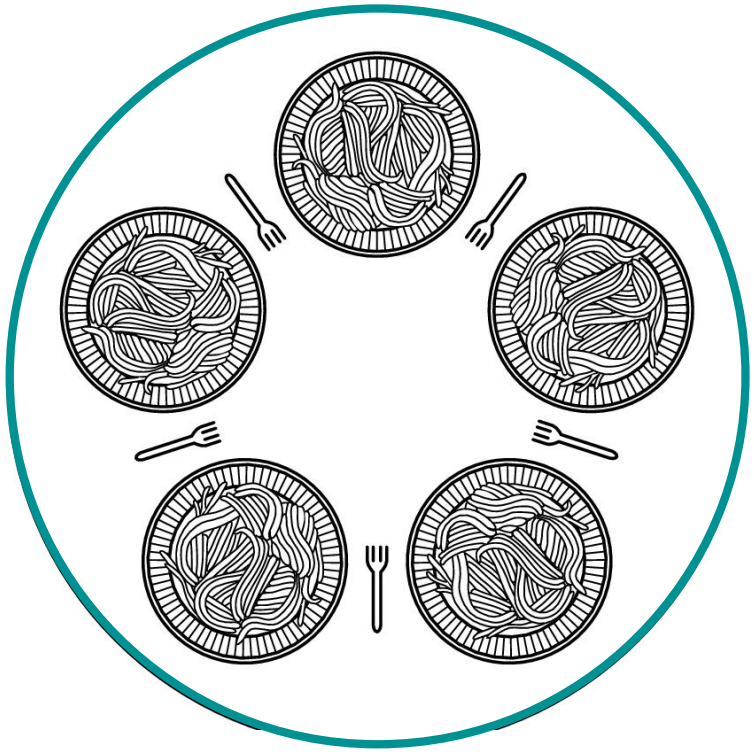
Jantar dos Filósofos



- Cada filósofo possui um prato de espaguete
- Para comer o espaguete o filósofo precisa de dois garfos
- Existe um garfo entre cada par de pratos
- Um filósofo come ou medita
  - Quando medita não interage com seus colegas
  - Quando está com fome ele tenta pegar dois garfos um de cada vez. Ele não pode pegar um garfo que já esteja com outro filósofo
- Os garfos são os recursos compartilhados

Jantar dos Filósofos

- Um algoritmo deve estabelecer o ritual entre os filósofos tal que:
  - exclusão mutua seja garantida
    - entre os garfos comuns
  - *deadlock* seja evitado
  - *starvation* seja evitado



Jantar dos Filósofos

pode levar a deadlock

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
     while (TRUE) {
          think( );                             /* philosopher is thinking */
          take_fork(i);                         /* take left fork */
          take_fork((i+1) % N);                 /* take right fork; % is modulo operator */
          eat( );                               /* yum-yum, spaghetti */
          put_fork(i);                          /* put left fork back on the table */
          put_fork((i+1) % N);                  /* put right fork back on the table */
     }
}

void main(){
       parbegin ( philosopher (0), philosopher (1),
                  philosopher (2), philosopher (3),
                  philosopher (4), );
}
```

Jantar dos Filósofos

```
#define N              5                /* number of philosophers */
#define LEFT           (i+N−1)%N        /* number of i's left neighbor */
#define RIGHT          (i+1)%N          /* number of i's right neighbor */
#define THINKING       0                /* philosopher is thinking */
#define HUNGRY         1                /* philosopher is trying to get forks */
#define EATING         2                /* philosopher is eating */
typedef int semaphore;                  /* semaphores are a special kind of int */
int state[N];                           /* array to keep track of everyone's state */
semaphore mutex = 1;                    /* mutual exclusion for critical regions */
semaphore s[N];                         /* one semaphore per philosopher */

void philosopher(int i)                 /* i: philosopher number, from 0 to N−1 */
{
      while (TRUE) {                    /* repeat forever */
            think( );                   /* philosopher is thinking */
            take_forks(i);              /* acquire two forks or block */
            eat( );                     /* yum-yum, spaghetti */
            put_forks(i);               /* put both forks back on table */
      }
}
```

Jantar dos Filósofos

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

se conseguiu um garfo, está com fome

se não conseguiu, s[i] está zero e fica bloqueado.

consegue o outro garfo, dependendo dos vizinhos

se conseguiu, s[i] vai para 1

Jantar dos Filósofos

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

id do filósofo esquerdo e direito

Pensar

1. Três processos compartilham quatro unidades de recursos que podem ser alocados e liberados um por unidade de tempo. Cada processo precisa de no máximo duas unidades de recursos. Mostre que deadlock não ocorrerá.

2. Comente sobre a seguinte solução para o problema do jantar dos filósofos:

   - um filósofo com fome, pega primeiramente o garfo da esquerda;
   - se o garfo da direita está disponível, ele pega o garfo e come
   - senão, ele devolve o garfo da esquerda e inicia o ciclo outra vez