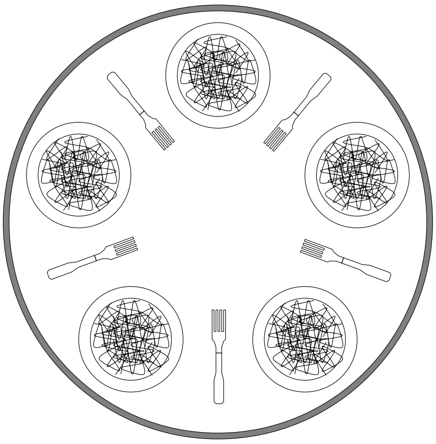
2.4 - Concorrência: Deadlock e Starvation

- nesta seção discutimos os princípios que suportam o *deadlock*; problemas relacionados com a condição de *starvation* e maneiras de tratar um *deadlock:* prevenindo, detectando ou evitando (*avoidance*);

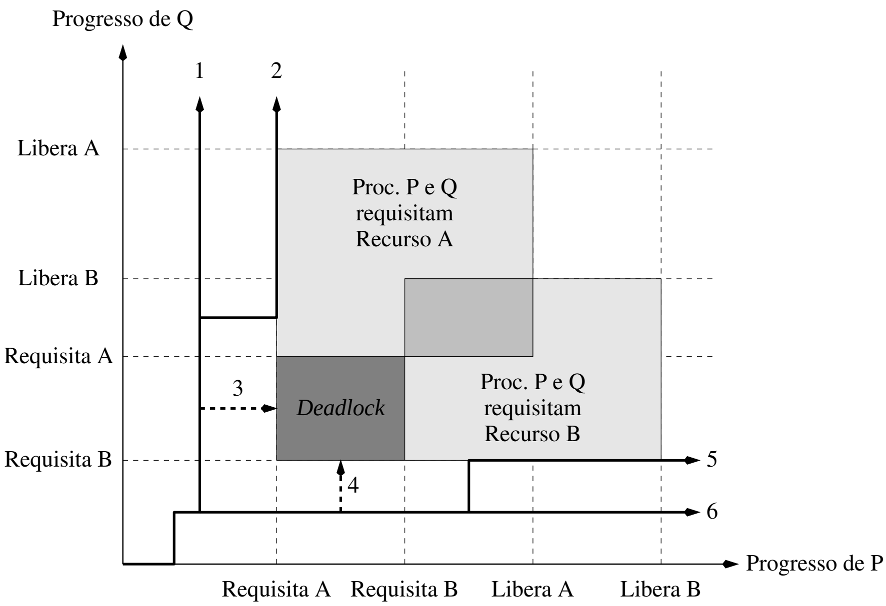
- por fim, ilustramos aspectos do *deadlock* e da sincronização entre processos utilizando o problema clássico dos filósofos jantando.



- o ciclo de vida de um filósofo consiste em períodos alternados de comer e pensar;
- quando um filósofo deseja comer, ele tenta adquirir um garfo a sua direita e um a sua esquerda, um por vez num dado momento.
- se bem sucedido na aquisição dos garfos, o filósofo se põe a comer por um momento, recoloca os garfos na mesa e começa a pensar.
- Questão: você poderia escrever um programa que simule cada filósofo no que é esperado que ele faça sem que o mesmo fique parado ?

2.4.1 - Concorrência: Deadlock

- **definição de deadlock:** bloqueio permanente de um conjunto de processos que competem por recursos do sistema ou comunicam-se uns com os outros.



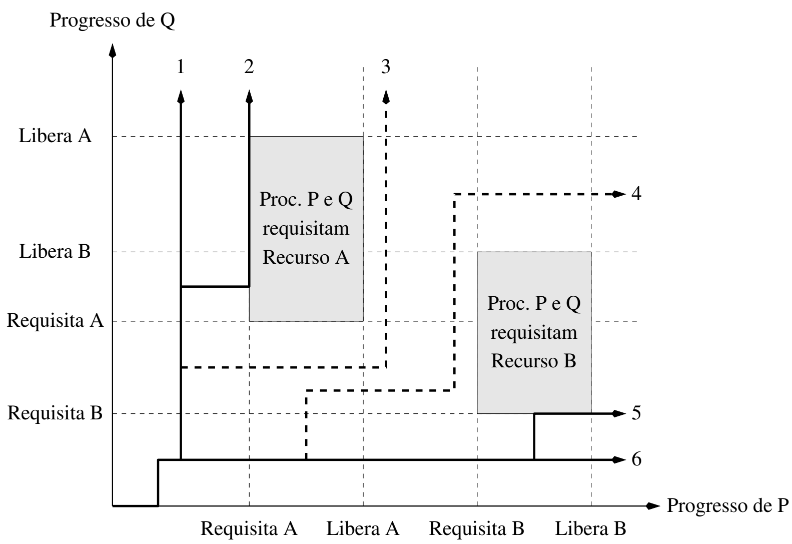
... 2.4.1 - Concorrência: Deadlock

- ... como apresentado na figura anterior, descrevemos a seguir as 06 possibilidades apresentadas, no que se refere a alocação dos Recursos A e B pelos Processos P e Q:

  1. Q aloca B e depois A, e então libera B e A. Quando P reassume a execução, ele irá ser capaz de alocar ambos os recursos.
  2. Q aloca B e depois A. P executa e bloqueia quando solicita A. Q libera B e A. Quando P reassume a execução, ele será capaz de alocar ambos os recursos.
  3. Q aloca B e então P aloca A. O $deadlock$ será inevitável, dado que a execução de ambos irá prosseguir, Q irá bloquear em A e P irá bloquear em B.
  4. P aloca A e então Q aloca B. O $deadlock$ será inevitável, dado que no prosseguimento da execução, Q irá bloquear em A e P irá bloquear em B.
  5. P aloca A e depois B. Q executa e bloqueia na requisição de B. P libera A e B. Quando Q reassume a execução, ele irá ser capaz de adquirir ambos os recursos.
  6. P aloca A e depois B e então libera A e B. Quando Q reassume a execução, ele poderá alocar ambos os recursos.

... 2.4.1 - Concorrência: Deadlock

- Se o $deadlock$ ocorre ou não, isto depende da dinâmica de execução e dos detalhes da aplicação, por exemplo: suponha no exercício anterior que P não necessite de ambos os recursos ao mesmo tempo conforme ilustrado abaixo.



... 2.4.1 - Concorrência: Deadlock

- ... duas categorias gerais de recursos podem ser distinguidas:
  1. reutilizáveis - recurso que pode seguramente ser usado por um processo em algum momento e não será exaurido pelo uso, permitindo assim uso posterior;
  2. consumíveis - são recursos que podem ser criados e destruídos; tipicamente, não há limite no número de recursos consumíveis de um tipo em particular.

- Como exemplo de *deadlock* involvendo recursos reutilizáveis, considere o cenário estabelecido entre 02 processos que estão competindo por recursos: ... neste cenário, que seqüência (*p0,q0,p1,...*) cria um *deadlock* !?

Steps	*Process P*	*Steps*	*Process Q*
	...		...
p0	*request(D);*	*q0*	*request(T);*
p1	*lock(D);*	*q1*	*lock(T);*
p2	*request(T);*	*q2*	*request(D);*
p3	*lock(T);*	*q3*	*lock(D);*
p4	*... perform action ...*	*q4*	*... perform action ...*
p5	*unlock(D);*	*q5*	*unlock(T);*
p6	*unlock(T);*	*q6*	*unlock(D);*

... 2.4.1 - Concorrência: Deadlock

- Como exemplo de *deadlock* involvendo recursos consumíveis, considere o cenário estabelecido entre 02 processos que estão comunicando um com o outro:

Steps	*Process P1*	*Steps*	*Process P2*
	...		...
p0	*receive(P2);*	*q0*	*receive(P1);*
	...		...
p1	*send(P2);*	*q1*	*send(P1);*
	...		...

- ... para o cenário anterior, a seqüência (*p0,q0,p1,...*) de execução é o que cria um *deadlock* !? ... há outra alternativa para termos um *deadlock* !?

2.4.2 - Condições para Ocorrência de Deadlock

- Três condições devem estar presentes para que um *deadlock* seja possível:

  ❶ **exclusão mútua** - somente um processo pode utilizar um dado recurso por vez;
  ❷ **manter e esperar** - um processo pode manter recursos alocados enquanto espera pela alocação de outros recursos;
  ❸ **nenhuma preempção** - nenhum recurso pode ser retirado de maneira forçada de um processo que o alocou e que o tenha mantido.

- *Deadlock* pode existir com estas 03 condições, mas não necessariamente com apenas estas 03 condições; ... uma quarta condição se faz necessária:

  ❹ **espera circular** - uma cadeia fechada de processos em estado de espera deve existir, ou seja, cada processo mantém ao menos um recurso que está sendo requisitado pelo próximo processo na cadeia de processos.

- ... as 03 primeiras condições são necessárias, mas não suficientes para que um *deadlock* ocorra — a quarta condição é de fato uma conseqüência potencial das três primeiras, ou seja, pode ou não acontecer.

2.4.3 - Prevenção, Evitação e Detecção de Deadlock

✯ **estratégia de prevenção** - consiste em projetar o sistema de modo que a possibilidade de *deadlock* seja excluída *a priori*;

- ... métodos de prevenção podem ser enquadrados em duas classes:

  ❶ método indireto de prevenção - prevenir a ocorrência de uma das 03 condições necessárias descritas anteriormente, ou seja, condições ❶, ❷ e ❸.
  ❷ médodo direto de prevenção - prevenir a ocorrência da espera circular (Cond. ❹).

✯ **estratégia de evitação** - embora pareça um pouco confuso (*avoidance*), esta estratégia permite a ocorrência das 03 condições necessárias, mas faz escolhas sensatas que garantam que o *deadlock* não será atingido.

- ... permitindo mais concorrência do que prevenção de concorrência, esta estratégia toma decisões dinamicamente com base no efeito que as requisições de alocações correntes terão na condução de um *deadlock*.

... 2.4.3 - Prevenção, Evitação e Detecção de Deadlock

- ... uma visão abstrata da lógica de evitar *deadlock* é apresentada a seguir:

```
type state = record
   resource, available: array[ 0 .. 1 ] of integer;
   claim, allocated: array[ 0 .. n-1, 0 .. m-1 ] of integer;
end;

if alloc[ i, * ] + request[ * ] > claim[ i, * ] then . . . error . . .      /* total request > claim */
else
   if request[ * ] > available[ * ] then . . . suspend process . . .
   else                                                                    /* simulate allocation */
      . . . define newstate by:
      begin
         allocation[ i, * ] := allocation[ i, * ] + request[ * ];
         available[ * ] := available[ * ] - request[ * ];
      end;
   end;

   if safe( newstate ) then
      . . . carry out allocation . . .
   else
      . . . restore original state . . .
      . . . suspend process . . .
   end;
end.
```

Resource Allocation Algorithm

... 2.4.3 - Prevenção, Evitação e Detecção de Deadlock

- ... como os recursos são atribuídos por tentativa ao processo i, um teste de segurança deve ser realizado para verificar se o estado para o qual o processo é levado é seguro.

```
type state = record
   resource, available: array[ 0 .. 1 ] of integer;
   claim, allocated: array[ 0 .. n-1, 0 .. m-1 ] of integer;
end;

function safe( state: S ): boolean;
   var current_avail: array[ 0 . . 1 ] of integer;
   rest: set of process;
begin
   current_avail := available;
   rest := { all processes };
   possible := true;
   while possible do
      find a Pk in rest such that
         claim[ k, * ] - alloc[ k, *] <= current_avail;
      if found then
         current_avail := current_avail + allocation[ k, * ];
         rest := rest - { Pk };
      else
         possible := false;
      end;
   end;
   safe := ( rest = null );
end.
```

Test for Safety Algorithm
(Banker's Algorithm)

... 2.4.3 - Prevenção, Evitação e Detecção de Deadlock

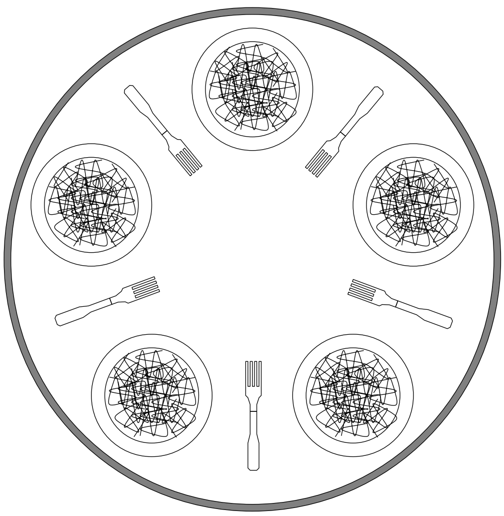
✯ **estratégia de detecção** - não limitando o acesso aos recursos ou restringindo as ações de processos, periodicamente, o sistema operacional executa um algoritmo que o permite detectar a condição de espera circular (Condição ❹).

- ... uma vez que o *deadlock* tenha sido detectado, algumas estratégias fazem-se necessárias para recuperá-lo do estado de bloqueio:

  ❶ abandona todos os processos que pertencem ao *deadlock*;
  ❷ recupera cada processo com *deadlock* para algum estado previamente definido de *checkpoint* e reinicia todos os processos;
  ❸ abandone sucessivamente processos (p. ex., ordem com base a algum critério que minimize custos) com *deadlock* até que o *deadlock* não mais exista.
  ❹ *preempção* sucessiva de recursos até que o *deadlock* não mais exista; como na estratégia anterior, uma seleção baseada em custo deve ser usada.

2.4.4 - Problema dos Filósofos Jantando

- 05 filósofos estão sentados ao redor de uma mesa, cada um com um prato contendo espaghetti muito fino, de modo que exige que cada filósofo para comer utilize dois garfos, um à sua direita e outro à sua esquerda.



- o ciclo de vida de um filósofo consiste em períodos alternados de comer e pensar;
- quando um filósofo deseja comer, ele tenta adquirir um garfo a sua direita e um a sua esquerda, um por vez num dado momento.
- se bem sucedido na aquisição dos garfos, o filósofo se põe a comer por um momento, recoloca os garfos na mesa e começa a pensar.
- Questão: você poderia escrever um programa que simule cada filósofo no que é esperado que ele faça sem que o mesmo fique parado ?

... 2.4.4 - Problema dos Filósofos Jantando

- Nesta solução, cada filósofo pega inicialmente o garfo da esquerda e depois o da direita. Após terminar de comer, os 02 garfos são recolocados na mesa.
- ... esta solução no entanto é passível de *deadlock* !

```
program dining_philosophers;
   var fork: array[ 0 .. 4 ] of semaphore (:=1);
   i: integer;

procedure philosopher( i: integer );
begin
   repeat
      think;
      wait( fork[ i ] );
      wait( fork[ (i+1) mod 5 ] );
      eat;
      signal( fork[ (i+1) mod 5 ] );
      signal( fork[ i ] );
   forever
end;

/* Main Program */
begin
   parbegin
      philosopher( 0 );
      philosopher( 1 );
      philosopher( 2 );
      philosopher( 3 );
      philosopher( 4 );
   parend;
end.
```


... 2.4.4 - Problema dos Filósofos Jantando

- ... uma solução é considerar um atendente que somente permite 4 filósofos num mesmo instante na mesa, assim pelo menos um filósofo poderá comer.

```
program dining_philosophers;
   var fork: array[ 0 .. 4 ] of semaphore (:=1);
   room: semaphore (:=4);
   i: integer;

procedure philosopher( i: integer );
begin
   repeat
      think;
      wait( room );
      wait( fork[ i ] );
      wait( fork[ (i+1) mod 5 ] );
      eat;
      signal( fork[ (i+1) mod 5 ] );
      signal( fork[ i ] );
      signal( room );
   forever
end;

/* Main Program */
begin
   parbegin
      philosopher( 0 );
      philosopher( 1 );
      philosopher( 2 );
      philosopher( 3 );
      philosopher( 4 );
   parend;
end.
```