Cap. 6 - Deadlock and Starvation

... Cap. 6 - Deadlock and Starvation

✯ William STALLINGS; **Operating Systems: Internals and Design Principles**, New Jerssey, Prentice-Hall, 1998, ISBN: 0-13-887407-7

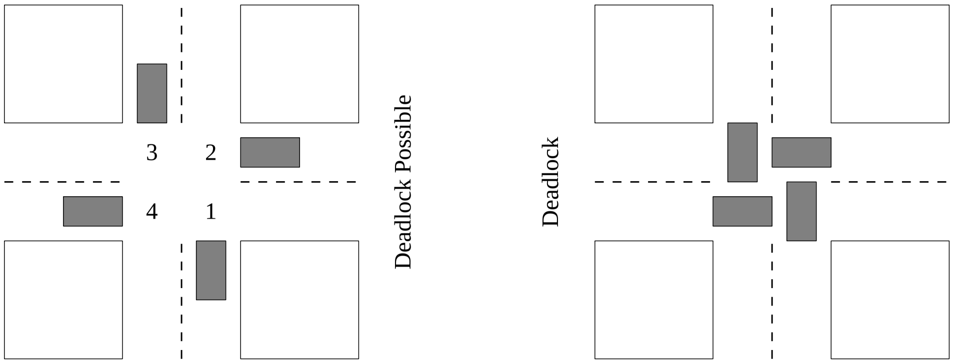
✯ Eleri CARDOZO; Maurício MAGALHÃES; Luís F. FAINA; **Sistemas Operacionais**, Dep. de Eng. de Computação e Automação Industrial, Fac. de Engenharia Elétrica e de Computação, UNICAMP, 1996.

6.1 - Princípios do Deadlock

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

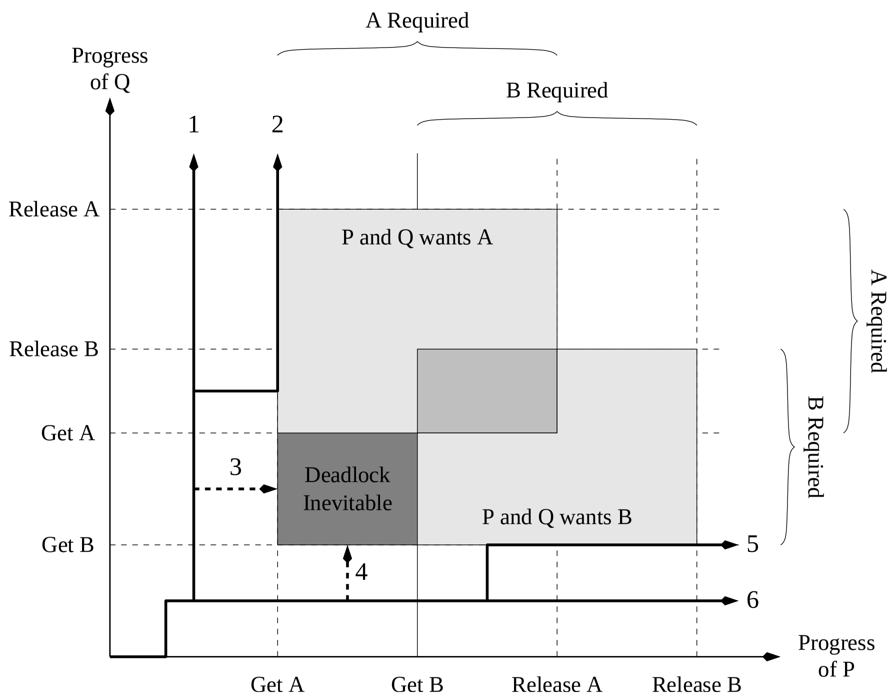
- ... diferentemente de outros problemas no gerenciamento de concorrência entre processos, não há solução eficiente para o *deadlock*.

- ... toda a ocorrência de *deadlock* envolve necessidades conflitantes de recursos por 02 ou mais processos, p.ex.: bloqueio de carros em um cruzamento.



... 6.1 - Princípios do Deadlock

✰ Representação de um *deadlock* envolvendo processos e recursos computacionais:



... 6.1 - Princípios do 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:

❶ 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.

❷ 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.

❸ 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.

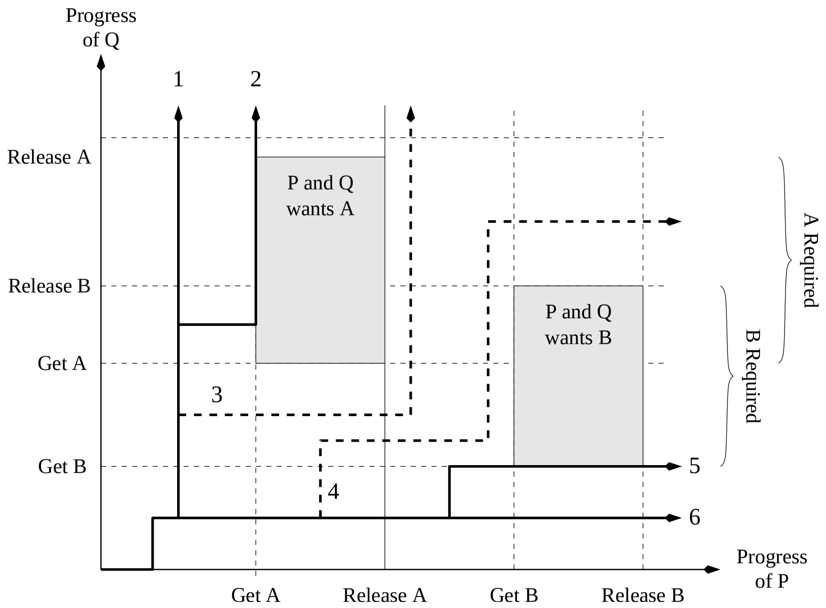
❹ P aloca A e então Q aloca B. O *deadlock* será inevitável, dado que no prosseguimento da execução, Q irá bloquear em A e P irá bloquear em B.

❺ 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.

❻ P aloca A e depois B e então libera A e B. Quando Q reassume a execução, ele poderá alocar ambos os recursos.

... 6.1 - Princípios do 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.



... 6.1 - Princípios do Deadlock

✠ Duas categorias gerais de recursos podem ser distinguidas:

❶ reutilizáveis - recurso que pode seguramente ser usado por um processo em algum momento e não será exaurido pelo uso, permitindo assim uso posterior;

❷ 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* envolvendo 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);**

... 6.1 - Princípios do Deadlock

✠ Como exemplo de *deadlock* envolvendo 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* !?

... 6.1 - Princípios do 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.

... 6.1 - Princípios do Deadlock

✡ *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.

6.2 - Deadlock Prevention

✰ **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:

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

❶ Geralmente, a primeira condição não pode ser desabilitada, posto que o acesso para estes recursos já pressupõe garantia de exclusão mútua.

❷ Por esta condição poderíamos exigir que todos os processos requisitassem todos os recursos ao mesmo tempo, em vez de uma por vez com alguma eventual espera;

- ... tal proposta apresenta como desvantagens o fato de que o processo pode ter que esperar por um longo tempo ao mesmo tempo, recursos que já foram alocados serão igualmente mantido por um longo tempo.

... 6.2 - Deadlock Prevention

❸ Abordagem prática quando aplicada a recursos cujos estados podem facilmente serem salvaguardados e recuperados posteriormente, como no caso de um processador;

- p.ex.: se um processo requer um recurso que está sendo mantido por um outro processo, o sistema operacional pode preemptar o segundo processo e requerer a liberação dos recursos por parte deste processo.

❹ Para prevenir a condição de espera circular pode-se ordenar os tipos de recursos disponíveis, assim se um recurso R foi alocado a algum processo, o mesmo poderá alocar outros recursos desde que na seqüência seguinte ao do recurso R.

6.3 - Deadlock Avoidance

✰ **estratégia de avoidance** - 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*.

✰ Considere um sistema com n processos, m diferentes tipos de recursos e as seguintes definições acerca dos processos e recursos:

$Resource = (R_1, R_2, R_3, ..., R_m)$ total de recursos no sistema

$Available = (V_1, V_2, V_3, ..., V_m)$ total de recursos não alocados

... 6.3 - Deadlock Avoidance

- requerimentos de cada processo acerca de cada recurso:

$$Claim = \begin{bmatrix} C_{11} & C_{12} & ... & C_{1m} \\ C_{21} & C_{22} & ... & C_{2m} \\ C_{31} & C_{32} & ... & C_{3m} \\ ... & ... & ... & ... \\ C_{n1} & C_{n2} & ... & C_{nm} \end{bmatrix}$$

- alocação corrente dos recursos por cada um dos processos:

$$Allocation = \begin{bmatrix} A_{11} & A_{12} & ... & A_{1m} \\ A_{21} & A_{22} & ... & A_{2m} \\ A_{31} & A_{32} & ... & A_{3m} \\ ... & ... & ... & ... \\ A_{n1} & A_{n2} & ... & A_{nm} \end{bmatrix}$$

... 6.3 - Deadlock Avoidance

✰ Com base nos vetores e matrizes apresentados, algumas relações devem ser satisfeitas:

❶ os recursos ou estão alocados ou estão disponíveis: $R_i = V_i + \sum_{k=1}^{n} A_{ki}, \ \ \forall i$

❷ nenhum processo pode alocar mais recursos que os do sistema: $C_{ki} \leq R_i \ \ \forall k, i$

❸ nenhum processo pode alocar mais recursos de um dado tipo que o número originalmente solicitado: $A_{ki} \leq C_{ki}, \ \ \forall k, i$

✰ Posto tais premissas, podemos definir uma política de *deadlock avoidance* que se recusa a iniciar um novo processo se suas exigências levam ao *deadlock*:

$$R_i \geq C_{(n+1)i} + \sum_{k=1}^{n} C_{ki} \quad \forall i$$

... 6.3 - Deadlock Avoidance

✰ Também referenciada como *Banker's Algorithm*, esta estratégia foi originalmente proposta por Dijkstra em 1996 para um sistema número fixo de processos e recursos;

- **estado do sistema:** alocação corrente de recursos pelos processos;
- **estado seguro:** é o estado no qual pela menos uma seqüência possibilite que todos os processos completem sem levar ao *deadlock*.
- **estado inseguro:** estado que não garante nem mesmo uma seqüência que permita que todos os processos completem sua execução.

... 6.3 - Deadlock Avoidance

Initial State

Claim Matrix

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

Allocation Matrix

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

Available Vector

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

P2 runs to Completion

Claim Matrix

	R1	R2	R3
P1	3	2	2
P2	6	0	2
P3	3	1	4
P4	4	2	2

Allocation Matrix

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

Available Vector

R1	R2	R3
6	2	3

P1 runs to Completion

Claim Matrix

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

Allocation Matrix

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

Available Vector

R1	R2	R3
7	2	3

P3 runs to Completion

Claim Matrix

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

Allocation Matrix

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

Available Vector

R1	R2	R3
9	3	4

... 6.3 - Deadlock Avoidance

✠ Como determinar um estado inseguro?

Initial State

Claim Matrix

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

Allocation Matrix

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

Available Vector

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
1	1	2

P1 requests One Unit each of R1 and R3

Claim Matrix

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

Allocation Matrix

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

Available Vector

R1	R2	R3
0	1	1

... 6.3 - Deadlock Avoidance

- ... 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

... 6.3 - Deadlock Avoidance

- ... 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)

6.4 - Deadlock Detection

✰ **estratégia de detecção:** - não limitando o acesso aos recursos ou restringindo as ações de processos, periodicamente, o sistema operacional executa um algoritmo que 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.

... 6.4 - Deadlock Detection

Principles	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources.	Requesting all resource at once.	● Works well for processes that perform a single burst of activity. ● No preemption necessary.	● Inefficient ● Delays process initiation.
			● Convenient when applied to resources whose state can be saved and restored easily. ● Feasible to enforce via compile−time checks. ● Needs no run−time computation since problem is solved in system design.	● Preempts more often than necessary ● Preempts without much use. ● Disallows incremental resource requests.
Detection	Very liberal; requested resources are granted where to test for deadlock	Invoke periodically to test for deadlock.	● Never delays process initiation ● Facilitates online handling.	● Inherent preemption losses.
Avoidance	Selects midway between that of detection and prevention.	Manipulate to find at least one safe path.	● No preemption necessary.	● Future resource requirements must be known. ● Processes can be blocked for long periods.

6.5 - Estratégia Integrada de Deadlock

✰ Como observado na tabela anterior, encontramos vantagens e desvantagens em todas as estratégias, logo, pode ser mais eficiente utilizarmos cada estratégias no seu contexto em vez de escolhermos uma dentre elas, ou seja:

❶ agrupar recursos em classes;

❷ utilizar ordenação linear anteriormente definida para prevenção de espera circular, assim, preveni-se os *deadlocks* nas classes de recursos;

❸ dentro de uma mesma classe de recursos, utilizar o algoritmo mais apropriado para aquela classe, como forma de obter melhor eficiência.

... 6.5 - Estratégia Integrada de Deadlock

✰ Alguns exemplos do uso desta técnica:

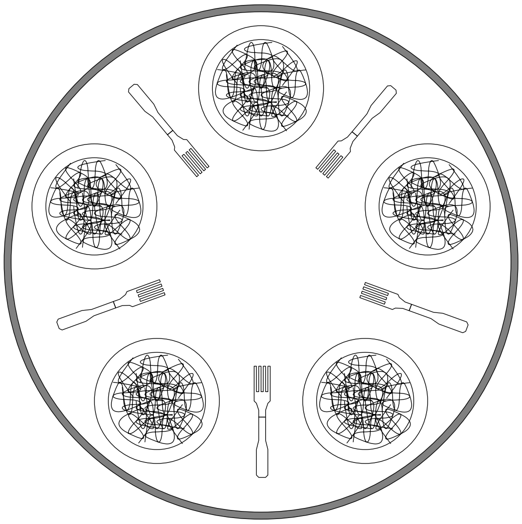
❶ **swappable space:** exigir que todos os pedidos de recursos sejam alocados ao mesmo tempo, ou seja, prevenção da condição *hold-and-wait*;

❷ **recuros de processos:** bastante efetiva nesta classe, a estratégia de *avoidance* exige que os processos declarem quais recursos e em que quantidade irão utilizar;

❸ **memória principal:** prevenção por preempção parece ser a estratégia mais apropriada neste caso, pois quando um processo é preemptado ele simplesmente é movido para a memória secundária.

6.6 - Problema dos Filósofos Jantando

✫ 05 filósofos estão sentados ao redor de uma mesa, cada um com um prato contendo spaghetti muito fino, de modo que exige que cada filósofo utilize dois garfos para comer, 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 ?**

... 6.6 - 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.
```

... 6.6 - Problema dos Filósofos Jantando

- ... uma solução é considerar um recepcionista 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.
```


... 6.6 - Problema dos Filósofos Jantando

✰ Solução para o Problema dos Filósofos Jantando proposta por TANENBAUM:

```
#define N   5                    /* number of philosophers */
#define LEFT    (i−1) mod N      /* number of i's left neighbor */
#define RIGHT     (i+1) mod 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 )        /* philosopher number, 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 */
   }
}

void take_forks( int i )         /* philosopher number, o to N−1 */
{
   wait( mutex );                /* enter critical region */
   state[i] = HUNGRY;            /* record fact that philosopher i is hungry */
   test( i );                    /* try to acquire 2 forks */
   signal( mutex );              /* exit critical region */
   wait( s[i] );                 /* block if forks were not acquired */
}

voit text( int i )
{
   if( state[i]==HUNGRY && state[LEFT]!=EATING
      && state[RIGHT] != EATING )  {
      state[i] = EATING;
      signal( s[i] );
   }
}

void put_forks( int i )          /* philosopher number, o to N−1 */
{
   wait( 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 */
   signal( mutex );              /* exit critical region */
}
```

6.7 -Mecanismos de Concorrência no UNIX

✰ Principais mecanismos de comunicação entre processos no UNIX SVR4:

❶ **pipes:** inspirado no conceito de co-rotina, é um *buffer* circular que possibilita a dois processos a comunicação segundo o modelo produtor-consumidor;

❷ **messages:** é um bloco composto de dados e tipo através do qual processos podem passar *msgsnd(...)* e receber *msgrcv(...)* mensagens;

❸ **shared memory:** forma mais rápida de comunicação entre processos, constitui-se de um bloco de memória compartilhado por múltiplos processos;

❹ **semaphores:** constituem generalizações das primitivas *wait(...)* e *signal(...)* apresentadas anteriormente e são executadas atomicamente pelo *kernel*;

❺ **signals:** mecanismo de *software* que informa ao processo a ocorrência de eventos assíncronos que, embora similar a interrupção de *hardware*, não emprega prioridades.

... 6.7 -Mecanismos de Concorrência no UNIX

✰ Além dos mecanismos de concorrência do UNIX SVR4, o Solaris suporta 04 primitivas de sincronização de *threads*: *mutual exclusion (mutex) locks*; *semaphores*; *multiple readers, single writer locks* e *condition variables*.

- ... tais primitivas são implementadas dentro do *kernel* através de *kernel threads* e disponibilizadas para o usuário por meio de uma biblioteca *user-level threads*.

- ... todas as primitivas de sincronização exigem a existência de instruções de *hardware* que possibilitam testar/atribuir um objeto em uma operação atômica.

... 6.7 -Mecanismos de Concorrência no UNIX

➊ **mutex locks:** previne que uma *thread* bloqueia o recurso associado quando o bloqueio já foi conseguido por outra *thread* através da primitiva *mutex_enter*;

- ... o tipo da ação no bloqueio depende de informações específicas armazenadas no objeto *mutex*, embora a política padrão seja *spin lock*;
- ... nesta política a *thread* bloqueada informa o *status* do bloqueio enquanto está executando o *loop* de espera, não obstante, pode se usar o mecanismo de bloqueio baseado em interrupção (anteriormente visto).

... 6.7 -Mecanismos de Concorrência no UNIX

➋ **semaphores:** são as seguintes as primitivas disponibilizadas:

- **sema_p(...):** decrementa o semáforo, potencialmente bloqueando a *thread*;
- **sema_v(...):** incrementa o semáforo, potencialmente desbloqueando a *thread*;
- **sema_tryp(...):** possibilita a espera ocupada caso em que decrementa o semáforo, posto que o mecanismo de bloqueio não foi configurado.

... 6.7 -Mecanismos de Concorrência no UNIX

❸ **readers/writer locks:** possibilita que múltiplas *threads* tenham acesso simultâneo de leitura em um objeto protegido pelo bloqueio;

- ... também permite que uma única *thread* acesse o objeto para escrita, *status* de *write lock*, enquanto todos os leitores estão bloqueados;

- ... todas as *threads* que tentarem o acesso para leitura ou escrita deverão esperar, no entanto, quando um o mais leitores adquiriram o *lock*, o *status* é de *read lock* (solução dos leitores com prioridade).

✰ **primitivas:** *rw_enter(...)*; *rw_exit(...)*; *rw_tryenter(...)*; *rw_downgrade(...)*; *rw_downgrade(...)*; *rw_tryupgrade(...)*;

... 6.7 -Mecanismos de Concorrência no UNIX

❹ **condition variables:** utilizada como unidade de espera até que uma condição em particular torne-se verdadeira, mas necessitam ser utilizadas com *mutex lock*.

- ... implementam um monitor como o descrito anteriormente, com as seguintes primitivas: *cv_wait(...)*; *cv_signal(...)*; *cv_broadcast(...)*;

6.8 - Mecanismos de Concorrência no Windows NT

✢ O mecanismo utilizado pelo *NT Executive* para implementar as facilidades de sincronização é uma família de objetos de sincronização, constituída de:

- ... processos; *threads*; arquivos; *console*; notificações de mudança de arquivo; *mutex*; *semaphores*, eventos e *waitable timers*.

✢ Os 04 primeiros, embora não são especialmente projetados para sincronização, podem ser utilizados para sincronização e comunicação.

- ... cada instância do objeto de sincronização pode se encontrar no estado de sinalizado ou de não sinalizado, sendo que a *thread* poderá ser suspensa somente quando o objeto encontra-se no estado de não sinalizado;

- ... quando um objeto entra no estado de sinalizado, o *NT Executive* libera todos os objetos *threads* que estejam esperando naquele objeto de sincronização.

... 6.8 - Mecanismos de Concorrência no Windows NT

Object Type	Definition	Set to Signaled State When	Effect on Waiting Threads
Process	A program invocation, including the address space and resources required to run the program.	Last thread terminates.	All released.
Thread	An executable entity within a process.	Thread terminates.	All released.
File	An instance of an opened file or I/O device.	I/O operation completes.	All released.
Console Input	A text window screen buffer. (e.g., used to handle screen I/O for an MS−DOS application)	Input is available for processing.	One thread released.
File Change Notification	An notification of any file system changes.	Change occurs in file system that matches filter criteria of this object.	One thread released.
Mutex	A mechanism that provides mutual exclusion cababilities for the Win32 and OS/ environments.	Owning thread or other thread releases the mutant.	One thread released.
Semaphore	A counter that regulates the number of threads that can use a resource.	Semaphore count drops to zero.	All released.
Event	An announcement that a system event has occured.	Thread sets the event.	All released.
Waitable Timer	A counter that records the passage of time.	Set time arrives or time interval expires.	All released.