... 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.
```

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

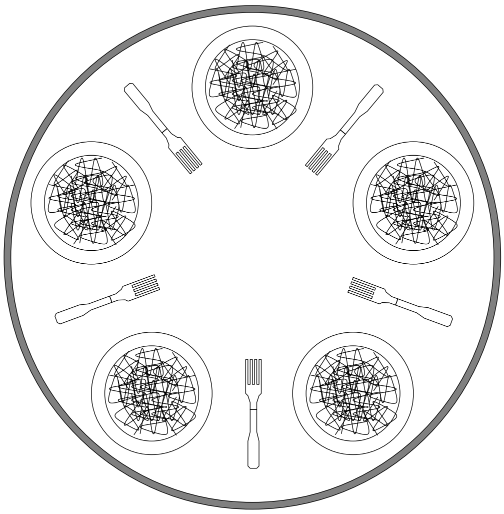
✯ **estratégia de detecção** - não limitando o acesso aos recursos ou restringindo as ações de processos, periodicamente, o sistema operacional executa um algoritmo que o permite detectar a condição de espera circular (Condição ❹).

- ... 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 ?

... 2.4.3 - 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)

2.4.2 - 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*.

... 2.4.1 - 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* !?

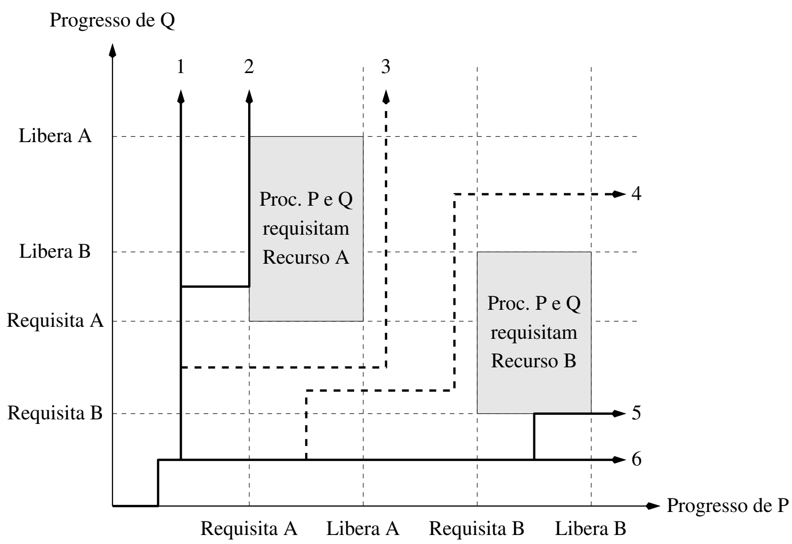
... 2.4.1 - Concorrência: Deadlock

- ... 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

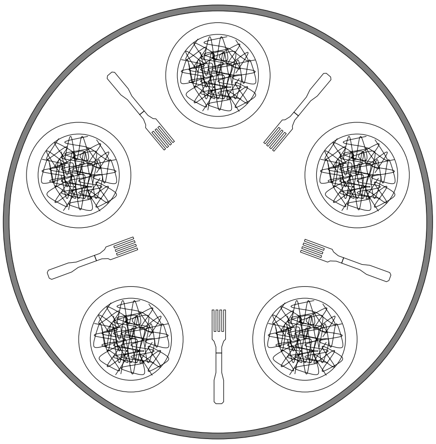
- Se o *deadlock* ocorre ou não, isto depende da dinâmica de execução e dos detalhes da aplicação, por exemplo: suponha no exercício anterior que P não necessite de ambos os recursos ao mesmo tempo conforme ilustrado abaixo.



2.4 - 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*);

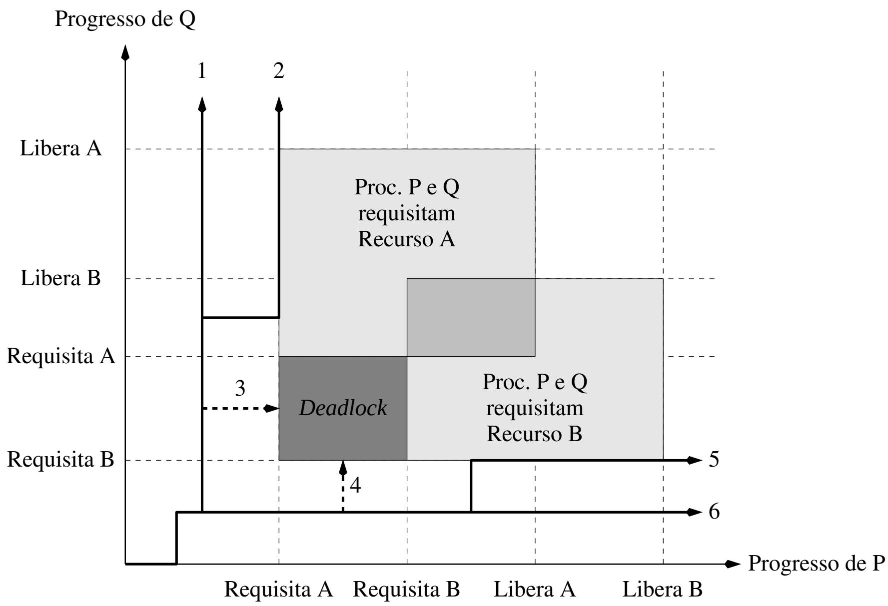
- por fim, ilustramos aspectos do *deadlock* e da sincronização entre processos utilizando o problema clássico dos filósofos jantando.



- 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.



... 2.3.8 - 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.
```


... 2.3.8 - 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.
```

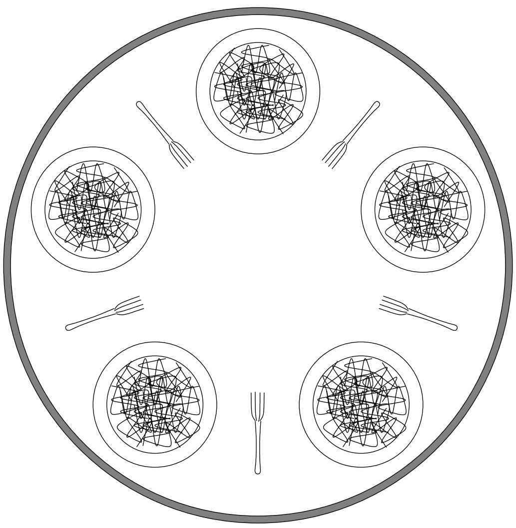
... 2.3.7 - Problema dos Leitores/Escritores

- ... esta solução considera o contexto em que os escritores têm prioridade e isto acarreta a adição de mais semáforos: $rsem$ para inibir leitores enquanto há algum escritor em atividade; $writecount$ que controla a atribuição de $rsem$ e o semáforo y que controla a atualização de $writecount$.

```
program readers_and_writers;
   var readcount, writecount: integer;
   x, y, z, wsem, rsem: semaphore (:=1);

procedure reader;
begin
   repeat
      wait( z );
         wait( rsem );
            wait( x );
               readcount := readcount + 1;
               if readcount = 1 then wait(wsem);
            signal(x)
         signal( rsem );
      signal( z );
      READ_UNIT;
      wait( x );
         readcount := readcount - 1;
         if readcount = 0 then signal( wsem );
      signal( x );
   forever;
end;

procedure writer;
begin
   repeat
      wait( y );
         writecount := writecount - 1;
         if writecount = 1 then wait( rsem );
      signal( y );
      wait( wsem );
      WRITE_UNIT;
      signal( wsem );
      wait( y );
         writecount := writecount -1;
         if writecount = 0 then signal( rsem );
      signal( y );
   forever;
end;

begin      /* Main Program */
   readcount := 0;
   parbegin
      reader;   writer;
   parend;
end.
```


2.3.8 - 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 ?

2.3.7 - Problema dos Leitores/Escritores

- caracterização: há uma área compartilhada (p. ex., bloco da memória principal, um arquivo ou até mesmo um banco de registradores de processadores) entre um número de processos, alguns deles lendo e outros apenas escrevendo na área de dados.
- ... as seguintes condições devem ser satisfeitas:

❶ qualquer número de leitores pode simultaneamente ler o arquivo;
❷ somente um escritor por vez pode escrever no arquivo;
❸ se um escritor está escrevendo no arquivo, nenhum leitor pode lê-lo.

- !? podemos considerar o problema produtor/consumidor como um caso especial do problema dos leitores/escritores com um único escritor (produtor) e um único leitor (consumidor) !? ... justifique ...
- ... duas soluções serão analisadas para este problema:

❶ leitores têm prioridade;
❷ escritores têm prioridade.

... 2.3.7 - Problema dos Leitores/Escritores

- ... a solução a seguir descreve uma instância para cada leitor e escritor do problema dos leitores/escritores e faz uso dos semáforos para garantir a exclusão mútua.
- nesta solução, os leitores têm prioridade, ou seja, é possível que os leitores mantenham controle da área de dados enquanto um leitor estiver lendo.

```
program readers_and_writers;
    var readcount: integer;
    x, wsem: semaphore (:=1);

procedure reader;
begin
    repeat
        wait(x);
            readcount := readcount + 1;
            if readcount = 1 then wait(wsem);
        signal(x)
        READ_UNIT;
        wait(x);
            readcount := readcount - 1;
            if readcount = 0 then signal(wsem);
        signal(x);
    forever;
end;

procedure writer;
begin
    repeat
        wait(wsem);
            WRITE_UNIT;
        signal(wsem);
    forever;
end;

/* Main Program */
begin
    readcount := 0;
    parbegin
        reader;
        writer;
    parend;
end.
```

2.3.6 - Mecanismo de Passagem de Mensagem

- quando processos interagem uns com os outros, 02 requisitos devem ser satisfeitos:
  - ✯ sincronização - exclusão mútua é garantida através da sincronização;
  - ✯ comunicação - cooperação de processos exige troca de informações.

- **passagem de mensagem** é uma abordagem que provê ambas funcionalidades e, adicionalmente, possibilita seu uso em sistemas distribuídos bem como em sistemas mono e multiprocessados com memória compartilhada.

- a abordagem passagem de mensagem pode aparecer em várias formas, mas normalmente é oferecida como um par de primitivas: *send(destination, message)* e *receive(source, message)*.

- ... este conjunto é o conjunto mínimo de operações necessárias para que processos possam utilizar a abordagem de passagem de mensagens.

... 2.3.6 - Mecanismo de Passagem de Mensagem

- como características de projeto em sistemas que empregam o mecanismo de passagem de mensagens na sincronização e comunicação entre processos, destacamos:
  - ❶ **sincronização:** *send* bloqueante e não bloqueante; *receive* bloqueante, não bloqueante, com teste para ver se há alguma mensagem esperando;
  - ❷ **endereçamento:** forma de especificar quem estará recebendo a mensagem, onde dois esquemas são possíveis: endereçamento direto e indireto;
  - ❸ **formato:** pode ser orientado a conteúdo ou atreladas ao tamanho, podendo serem mensagens de tamanho fixo ou variável;
  - ❹ **disciplina na fila de espera:** mensagens podem ser enfileiradas segundo uma lista FIFO (*First In First Out*), por prioridade, caso em que algumas mensagens são mais urgentes que outras ou permitir que o recebedor inspecione a fila de mensagens e selecione qual será a próxima mensagem a ser recebida.

... 2.3.5 - Mec. de Monitores - Prob. do Produtor/Consumidor

```
monitor  bounded_buffer;
   buffer: array[ 0 .. 1 ] of char;          /* space for N items */
   nextin, nextout: integer;                 /* buffer pointers */
   count: integer;                           /* number of items in buffer */
   notfull, notempty: condition;             /* for synchronization */

procedure  append( x: char );
begin
   if count = N then cwait(notfull);         /* buffer is full; avoid overflow */
   buffer[nextin] := x;
   nextin := nextin + 1 mod N;
   count := count + 1;                       /* one more item in buffer */
   csignal(notempty);                        /* resume any wait consumer */
end;

procedure  take( x: char );
begin
   if count = 0 then cwait(notempty);
   x := buffer[nextout];                     /* buffer is empty; avoid underflow */
   netxout := nextout + 1 mod N;
   count := count -1;                        /* one fewer item in buffer */
   csignal( notfull );                       /* resume any waiting producer */
end;

begin                                                /* Monitor Body */
   nextin := 0;  nextout := 0;  count := 0;          /* buffer initially empty */
end;
```


... 2.3.5 - Mec. de Monitores - Prob. do Produtor/Consumidor

```
procedure  producer;                         /* Procedure Producer */
var  x: char;
begin
   repeat
      produce(x);
      append(x);
   forever;
end;

procedure  consumer;                         /* Procedure Consumer */
var  x: char:
begin
   repeat
      take(x);
      consume(x);
   forever;
end;

begin                                        /* Main Program */
   parbegin
      producer; consumer;
   parend;
end.
```

... 2.3.5 - Mecanismo de Monitores

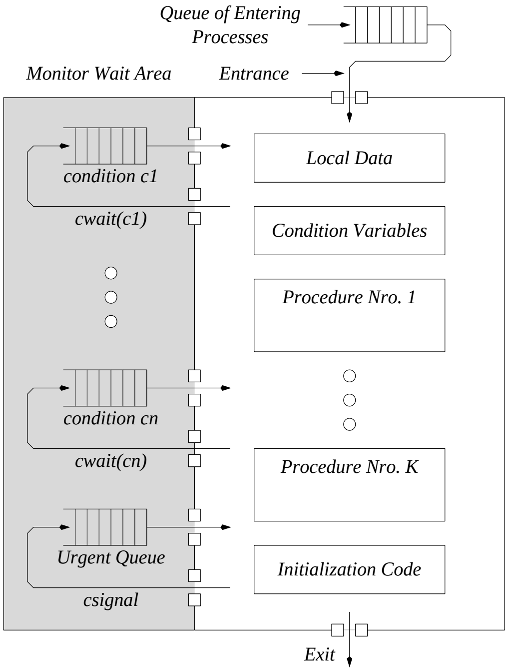
- monitor: é um módulo de *software* consistindo de um ou mais procedimentos, uma seqüência de inicialização e dados locais.

- constituem características do monitor:

  * os dados de variáveis locais são acessíveis somente pelos procedimentos do monitor;

  * um processo tem acesso ao monitor invocando um de seus procedimentos;

  * somente um processo pode executar um monitor em um dado instante, ou seja, qualquer outro processo que invocar o monitor será suspendido.



... 2.3.5 - Mecanismo de Monitores

- as duas primeiras características advém do conceito de objetos;

- com a disciplina de um processo por vez, o monitor garante a exclusão mútua;

- as variáveis no monitor podem ser acessadas por um único processo em um dado momento, desde modo, a estrutura de dados compartilhada pode ser protegida;

- o monitor oferece mecanismos de sincronização através das variáveis de condição e estando dentro do monitor são acessíveis somente no seu interior:
  - ✯ $cwait(c)$ - suspende a execução do processo que executa esta chamanda e torna o monitor disponível para uso por outro processo;
  - ✯ $csiganl(c)$ - recupera a execução de algum processo (p. ex., lista de processos suspensos) suspendido após o $cwait$ sobre alguma condição.

- note que as operações $wait/signal$ do monitor são diferentes daquelas apresentadas no semáforo - se um processo no monitor invoca uma operação de $signal$ e nenhuma tarefa está esperando aquela condição, o sinal é perdido.

... 2.3.4 - Mec. de Semáforos - Prob. do Produtor/Consumidor

- ... como mencionado, é imperativo que as operações *wait* e *signal* sejam atômicas, por exemplo, através de *hardware* ou *firmware*.

```
var  n: integer;
     n: semaphore( := 0 );
     s: semaphore( := 1 );

procedure producer:            procedure consumer:
begin                          begin
   repeat                         repeat
      produce;                       wait( n );
      wait( s );                     wait( s );
      append;                        take;
      signal( s );                   signal( s );
      signal( n );                   consume;
   forever                        forever
end;                           end;

begin  /* Main Program */
   n := 0;
   parbegin
      producer;    consumer;
   parend
end.
```


2.3.5 - Mecanismo de Monitores

- semáforos provêem uma solução flexível e forte, embora primitiva, para garantir a exclusão mútua e coordenação entre os processos;

- ... entretanto, pode ser difícil produzir um programa correto utilizando semáforos — esta dificuldade decorre do fato de que as operações *wait* e *signal* dispersas no programa dificultam a visualização do efeito (global) esperado no semáforo.

- **monitor:** é uma construção em linguagem de programação que provê funcionalidade equivalente a do semáforo, mas mais fácil de ser controlada.

- ... o conceito de monitores tem sido implementado em um grande conjunto de linguagens, incluindo: Pascal Concorrente, Pascal-plus, Modula-2 e Modula-3.

... 2.3.4 - Mec. de Semáforos - Prob. do Produtor/Consumidor

```
var  n: integer;
     s: binary_semaphore( := 1 );
     delay: binary_semaphore( := 1 );

procedure producer;
begin
   repeat
      produce;
      waitB( s );
      append;
      n := n + 1;
      if n = 1 then signalB( delay );
      signalB( s );
   forever
end;

procedure consumer;
begin
   waitB( delay );
   repeat
      waitB( s );
      take;
      n := n - 1;
      signalB( s );
      consume;
      if n = 0 then signalB( delay );
   forever
end;

begin  /* Main Program */
   n := 0;
   parbegin
      producer;    consumer;
   parend
end.
```


... 2.3.4 - Mec. de Semáforos - Prob. do Produtor/Consumidor

- ... há no entanto um erro neste programa - quando o consumidor consome todo o *buffer*, ele necessita reinicializar o semáforo *delay* de modo que ele possa ser forçado a esperar até que o produtor tenha colocado mais itens no *buffer*.

	Ação	*n*	*delay*
1	Inicialmente	0	0
2	Produtor: região crítica	1	1
3	Consumidor: waitB(delay)	1	0
4	Consumidor: região crítica	0	0
5	Produtor: região crítica	1	1
6	Consumidor: if n = 0 then waitB(delay)	1	1
7	Consumidor: região crítica	0	1
8	Consumidor: if n = 0 then waitB(delay)	0	0
9	Consumidor: região crítica	-1	0

... 2.3.4 - Mecanismo de Semáforos

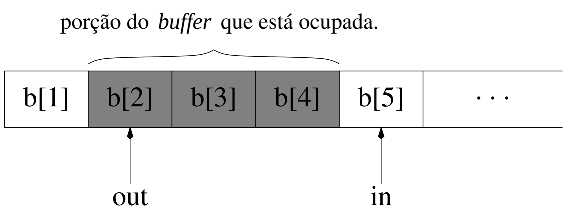
```
program  mutual_exclusion;

const n = ...;  /* number of processes */
var s: semaphore (:= 1);

procedure P( i: integer )
begin
  repeat
    wait( s );
    ... critical region ...
    signal( s );
    ... remainder ...
  forever;
end;

begin  /* Main Program */
  parbegin
    P( 1 );
    P( 2 );

    P( n );
  parend;
end.
```


... 2.3.4 - Mec. de Semáforos - Prob. do Produtor/Consumidor

- **problema do produtor/consumidor:** há um ou mais produtores gerando algum tipo de dado e armazenando-o em um *buffer* e um único consumidor retirando itens, um por vez, deste *buffer* e consumindo.

```
producer:
   repeat
      produce item v;
      b[in] := v;
      in := in + 1;
   forever;
```

```
consumer:
   repeat
      while in <= out do { nothing };
      w := b[out];
      out := out + 1;
      consume item w;
   forever;
```



... 2.3.4 - Mecanismo de Semáforos

```
type semaphore = record
    count: integer;
    queue: list of process;
  end;

var s: semaphore;

wait( s ):
  s.count := s.count -1;
  if s.count < 0
    then  begin
      place this process in s.queue;
      block this process;
    end;

signal( s ):
  s.count := s.count + 1;
  if s.count <= 0
    then  begin
      remove a process P from s.queue;
      place process P on ready list;
    end;
```


... 2.3.4 - Mecanismo de Semáforos

```
type binary_semaphore = record
    value: ( 0, 1 );
    queue: list of process;
  end;

var s: binary_semaphore;

waitB( s ):
  if s.value = 1;
    then  s.value := 0;
    else  begin
      place this process in s.queue;
      block this process;
    end;

signalB( s ):
  if s.queue is empty
    then  s.value := 1;
    else  begin
      remove a process P from s.queue;
      place process P on ready list;
    end;
```

... 2.3.3 - Exclusão Mútua: Solução por Hardware

- ... algumas propriedades da abordagem utilizando instruções de máquina:
  - ✔ é aplicável a um grande número de processos (p. ex., mono e multiprocessados).
  - ✔ a abordagem por instruções de máquina é simples e fácil de ser verificada.
  - ✔ pode ser usada para suportar múltiplas regiões críticas, com cada região crítica definindo sua própria variável.

- ... algumas desvantagens da abordagem utilizando instruções de máquina:
  - ✘ espera ocupda é empregada, o que gera consumo de processador;
  - ✘ possibilidade de ocorrer *starvation*, pois alguns dos processos esperando para entrar na região crítica podem ter que esperar indefinidamente;
  - ✘ possibilidade de *deadlock*, quando processos com prioridades diferentes estiverem um acessando a região crítica e o outro solicitando a região crítica.

2.3.4 - Mecanismo de Semáforos

- **idéia:** processos podem se comunicar por meio de sinais, de modo que um processo possa ser obrigado a parar em um dado ponto de seu processamento antes que tenha recebido um sinal em particular, assim, uma coordenação poderá ser estabelecida.

- **semáforo = variável especial** para sinalização - para transmitir um sinal via semáforo s, um processo executa a primitiva *signal(s)* e, para receber um sinal via semáforo o processo executa a primitiva *wait(s)*;

- ... a variável semáforo pode ser vista como uma variável que contém um inteiro e sobre a qual 03 operações são definidas:
  - ✯ o semáforo pode ser inicializado com valor negativo;
  - ✯ a operação *wait(s)* decrementa o valor do semáforo e caso o valor tenha se tornado negativo o processo executando a operação é bloqueado;
  - ✯ a operação *signal(s)* incrementa o valor do semáforo e testa se o valor é 0 ou negativo e em caso afirmativo um processo bloqueado é desbloqueado.

... 2.3.3 - Exclusão Mútua: Solução por Hardware

- **exchange instruction:** é uma operação atômica não sujeita a interrupções que permuta os conteúdos de uma registrador e de uma posição da memória.

```
procedure exchange  ( var r: register; var m: memory )
var  temp;
begin
  temp := m;
  m := r;
  r := temp;
end.
```


... 2.3.3 - Exclusão Mútua: Solução por Hardware

```
const n = ...;  /* number os processes */
var bolt: integer;

procedure P( i : integer );
var keyi: integer;
begin
  repeat
    keyi := 1;
    repeat exchange( keyi, bolt ) until keyi = 0;
    ... critial region ...
    exchange( keyi, bolt );
    ... remainder ...
  forever
end;

begin /* Main Program */
  bolt := 0;
  parbegin
    P(1);
    P(2);  ...  P(n);
  parend
end.
```

... 2.3.3 - Exclusão Mútua: Solução por Hardware

- ... solução alternativa: proposição de instruções de máquina que suportam ações de forma atômica, tais como: ler e escrever ou ler e testar uma posição da memória com um único ciclo de busca de instrução.

- **test and set instruction:** é uma operação atômica não sujeita a interrupções que testa o valor de seu argumento, modificando-o sob certa condição e retornando *true* ou *false* conforme a operação realizada.

```
function  testset  ( var i: integer ) : boolean;
begin
  if i = 0 then
    begin
      i := 1;
      testset := true;
    end;
  else  testset := false;
end.
```


... 2.3.3 - Exclusão Mútua: Solução por Hardware

```
const n = ...;  /* number os processes */
var bolt: integer;

procedure P( i : integer );
begin
  repeat
    repeat { nothing } until testset( bolt );
    ... critial region ...
    bolt := 0;
    ... remainder ...
  forever
end;

begin /* Main Program */
  bolt := 0;
  parbegin
    P(1);  P(2);  ...  P(n);
  parend
end.
```

... 2.3.3 - Solução Exclusão Mútua: Algoritmo de Peterson

```
var  flag:  array [0 .. 1] of boolean;
     turn: 0 .. 1;

Procedure P0;
begin
  repeat
    flag[0] := true;
    turn := 1;
    while flag[1] and turn = 1 do
      { nothing };
    ... critical region ...
    flag[0] := false;
    ... remainder ...
  forever;
end;

Procedure P1;
begin
  repeat
    flag[1] := true;
    turn := 0;
    while flag[0] and turn = 0 do
      { nothing };
    ... critical region ...
    flag[1] := false;
    ... remainder ...
  forever;
end;

/* Main Program - the same as before */
```


2.3.3 - Exclusão Mútua: Solução por Hardware

- ... do ponto de vista da exclusão mútua, um processo continuará sendo executado até que uma operação do sist. oper. seja requerida ou até ser interrompido;
- ... a solução mais simples é o método de desabilitar todas as interrupções quando se está entrando na região crítica e reabilitá-la ao sair.

```
repeat
  ... disable interrupts ...
  ... critical region ...
  ... enable interrupts ...
  ... remainder ...
forever
```

- ... pelo fato da região crítica não poder ser interrompida, a exclusão mútua está garantida, embora o preço pago por esta abordagem seja alto.

✘ a eficiência de execução pode notadamente ser degradada como conseqüência da limitação imposta ao processador no que tange a habilidade de entrelaçar processos.

... 2.3.3 - Solução Exclusão Mútua: Algoritmo de Dekker

```
Procedure P0;
begin
  repeat
    flag[0] := true;
    while flag[1] do if turn = 1 then
    begin
      flag[0] := false;
      while turn = 1 do { nothing };
      flag[0] := true;
    end;
    ... critical region ...
    turn := 1;
    flag[0] := false;
    ... remainder ...
  forever;
end;
```

```
Procedure P1;
begin
  repeat
    flag[1] := true;
    while flag[0] do if turn = 0 then
    begin
      flag[1] := false;
      while turn = 0 do { nothing }
      flag[1] := true;
    end;
    ... critical region ...
    turn := 0;
    flag[1] := false;
    ... remainder ...
  forever;
end;
```


... 2.3.3 - Solução Exclusão Mútua: Algoritmo de Dekker

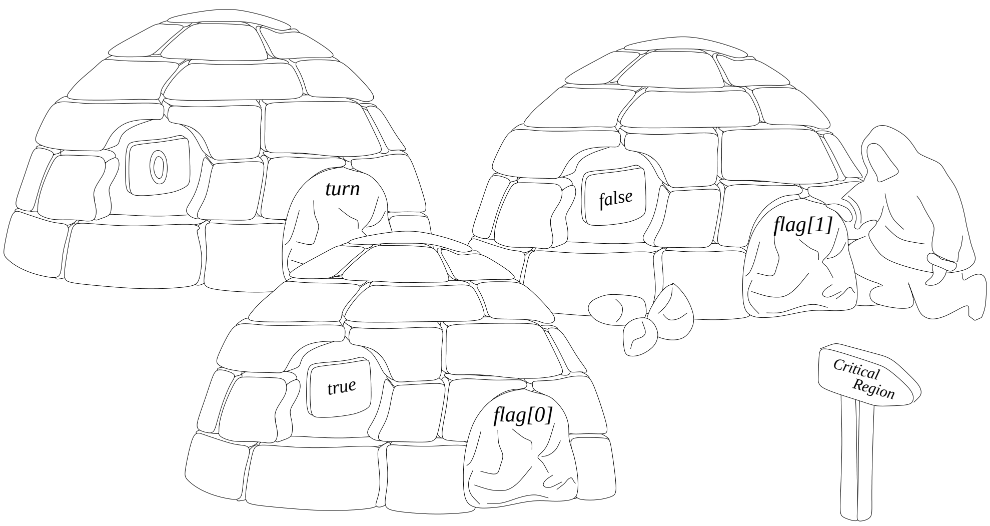
```
/* Main Program */
begin
  flag[0] := false;
  flag[1] := false;
  turn := 1;

  parbegin
    P0;  /* Procedure P0 */
    P1;  /* Procedure P1 */
  parend;
end;
```

... 2.3.3 - Solução Exclusão Mútua: Tentativa 4

```
var  flag:  array [0 .. 1] of boolean;

Process 0
  .
  .
  flag[0] := true;
  while flag[1] do
  begin
    flag[0] := false;
    ... delay for a short time ...
    flag[0] := true;
  end;
  ... critical region ...
  flag[0] := false;
  .
  .

Process 1;
.
  .
flag[1] := true;
while flag[0] do
begin
  flag[1] := false;
  ... delay for a short time ...
  flag[1] := true;
end;
... critical region ...
flag[1] := false;
.
.
```

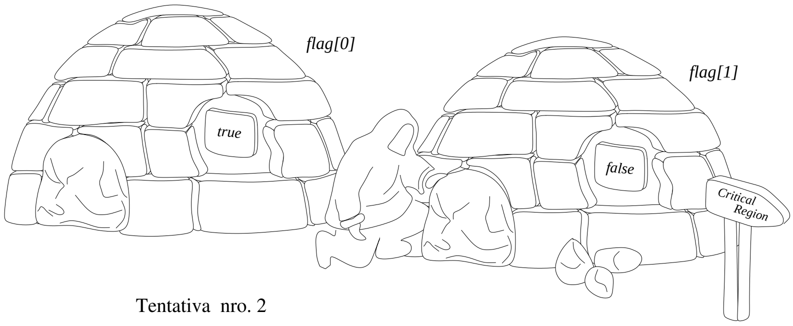

... 2.3.3 - Solução Exclusão Mútua: Algoritmo de Dekker

```
var  flag:  array [0 .. 1] of boolean;
     turn: 0 .. 1;
```



... 2.3.3 - Exclusão Mútua: Solução por Software

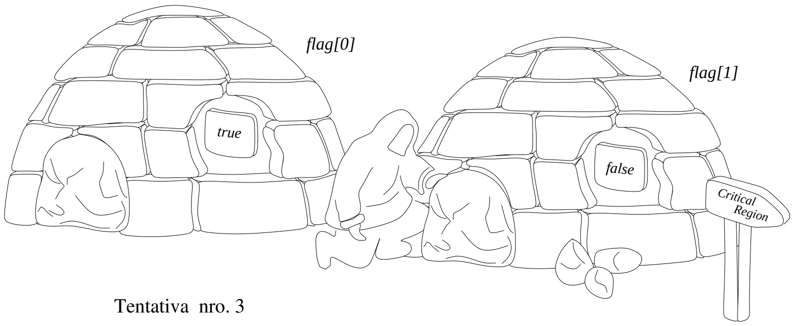
```
var  flag:  array [0 .. 1] of boolean;

Process 0                               Process 1
  .                                       .
  while flag[1] do { nothing; }         while flag[0] do { nothing }
  flag[0] := true;                      flag[1] := true;
  ... critical region ...               ... critical region ...
  flag[0] := false;                     flag[1] := false;
  .                                     .
  .                                     .
```



Tentativa nro. 2

... 2.3.3 - Exclusão Mútua: Solução por Software

```
var  flag:  array [0 .. 1] of boolean;

Process 0                               Process 1
  .                                       .
  flag[0] := true;                      flag[1] := true;
  while flag[1] do { nothing; }         while flag[0] do { nothing }
  ... critical region ...               ... critical region ...
  flag[0] := false;                     flag[1] := false;
  .                                     .
  .                                     .
```



Tentativa nro. 3

... 2.3.2 - Região Crítica e Exclusão Mútua

- um algoritmo de mútua exclusão possui invariavelmente duas partes:

  ❶ enter_critical - é evocada quando o processo deseja iniciar a execução de uma região crítica; ... ao retornar o processo está apto a executar a região crítica;
  ❷ exit_critical - é evocada para anunciar que a região crítica já foi executada.

```
program  mutual_exclusion;
const  n = ...;  /* number of processes */

procedure  P( i: integer );
begin
  repeat
    enter_critical( R );
    ... critical region ...
    exit_critical( R );
    ... ramainder ...
  forever
end;
```

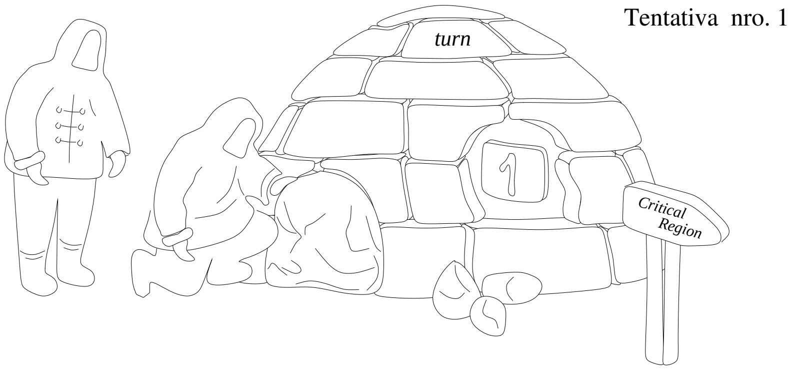
```
/* main program */
 begin
   parbegin
     P( 1 );
     P( 2 );
     ...
     P( n );
   parend
 end.
```


2.3.3 - Exclusão Mútua: Solução por Software

```
var  turn:  0 .. 1;
```

```
Process 0
  .
  while turn != 0 do { nothing; }
  ... critical region ...
  turn := 1;
  .
  .
```

```
Process 1
  .
  while turn != 1 do { nothing }
  ... critical region ...
  turn := 0;
  .
  .
```



2.3.2 - Região Crítica e Exclusão Mútua

- seja o exemplo abaixo que ilustra os elementos essenciais de um programa que "ecoa" em algum dispositivo de saída o caracter informado no dispositivo de entrada:

```
procedure echo;
var out, in: character;
begin
  input( in, keyboard );
  out := in;
  output( out, display );
end
```

- **região crítica:** a execução de uma região crítica (parte do código que manipula o recurso compartilhado) deve ser um procedimento controlado a fim de evitar que os recursos compartilhados atinjam estados inconsistentes.

- **solução:** não permitir que mais de um processo leia ou escreva dados compartilhados ao mesmo tempo, ou seja, deve-se garantir a **exclusão mútua**.

... 2.3.2 - Região Crítica e Exclusão Mútua

- ... embora a **mútua exclusão** evite inconsistências, o mecanismo não garante eficiência na utilização dos recursos compartilhados e, assim, para assegurarmos uma boa solução faz-se necessário que:

  ❶ 02 processos não estejam simultaneamente dentro de suas regiões críticas referentes ao mesmo recurso compartilhado;
  ❷ a mútua exclusão se dê de forma independente da velocidade relativa dos processos ou mesmo do número de processadores;
  ❸ nenhum processo executando fora da região crítica bloqueie outro processo;
  ❹ nenhum processo espere um tempo arbitrariamente longo para executar uma região crítica (p. ex., na ocorrência de um bloqueio temporário).

- para garantir as propriedades acima propostas, os algoritmos de controle são classificados segundo o modo que esperam pela autorização de entrada em uma região crítica: competindo pelo processador durante a espera ou bloqueado.

2.3 - Concorrência: Exclusão Mútua e Sincronização

- ... no projeto de sistemas operacionais, o tema central está associado ao gerenciamento de processos e *threads* nos mais diferentes áreas, tais como: multiprogramação, multiprocessamento e computação distribuída;

- **concorrência:** constitui-se num aspecto fundamental para todas essas áreas, bem como para o projeto de seus sistemas operacionais;

- o conceito de concorrência evidencia-se em 03 diferentes contextos:
  - aplicações múltiplas - a multiprogramação é resultado do compartilhamento do tempo de processamento entre aplicações ativas;
  - aplicações estruturadas - como uma extensão dos princípios da programação modular e estruturada, algumas aplicações podem ser efetivamente desenvolvidas como um conjunto de processos concorrentes;
  - estrutura do sistemas operacionais - as mesmas vantagens da estruturação aplicam-se aos programadores de sistemas e, pode ser verificado que os sistemas operacionais são implementados como um conjunto de processos.

2.3.1 - Princípios da Concorrência

- ... tanto em sistemas multiprogramados quanto nos multiprocessados, as técnicas de entrelaçamento e sobreposição podem ser vistas como exemplos de concorrência, e ambas apresentam os mesmos problemas:

  ✘ o compartilhamento de recursos globais está recheado de perigos - p. ex.: se 02 processos compartilham uma mesma variável global e ambos efetuam operações de escrita e leitura, então a ordem com que estas operações (escritas e leituras) ocorrem torna-se um elemento crítico;

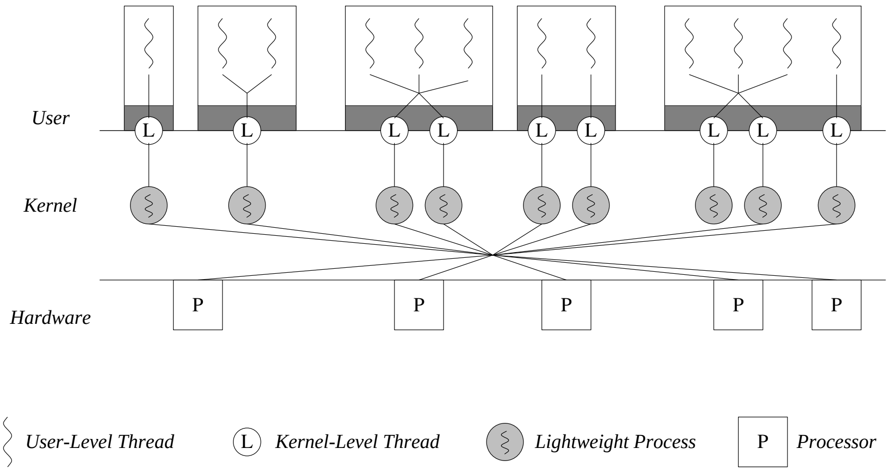
  ✘ é difícil para o sistema operacional gerenciar a alocação de recursos de forma ótima - exemplo: se um processo requisita o uso de um canal de I/O e em seguinda é suspenso, pode ser ineficiente para o S.O. simplesmente bloquear o canal e não permitir o seu uso por outros processos;

  ✘ torna-se difícil a localização de erros de programação, dado que os resultados são tipicamente não podem ser reproduzidos.

- razão para estas dificuldades: a execução de processos não pode ser predita pois esta depende das atividades de outros processos, da maneira pela qual as interrupções são manipuladas e das políticas de escalonamento.

... 2.2.4 - Threads no Solaris

- a combinação de *threads* no nível do *kernel* e do usuário oferecem ao programador de aplicação a oportunidade de explorar a concorrência de forma muito mais eficiente e apropriada para uma dada aplicação.



... 2.2.4 - Threads no Solaris

- *user-level threads* podem ser criadas, destruídas, bloqueadas, ativadas, etc. sem que o *kernel* seja envolvido;

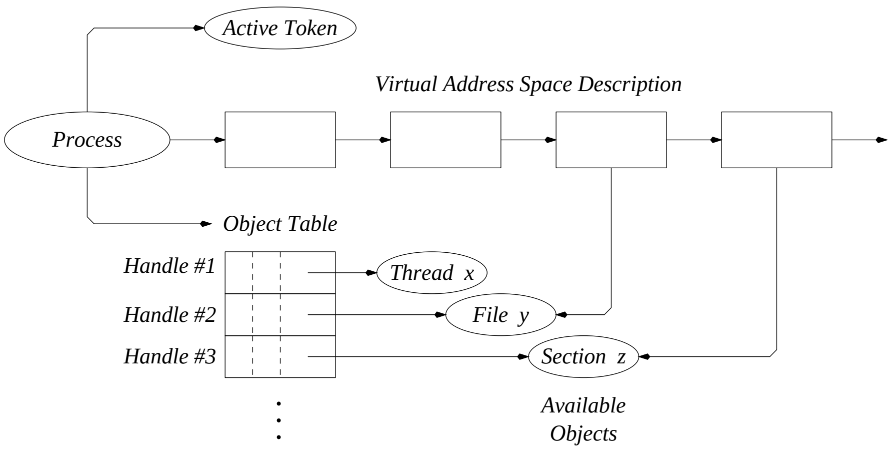
- ... quando uma *thread* do nível de usuário é conhecida pelo *kernel*, o *kernel* terá que alocar estruturas de dados do *kernel* para cada uma e realizar o chaveamento de *threads* — chaveamento no modo *kernel* é mais caro que no modo usuário;

- se uma aplicação involve *threads* que podem bloquear, tais como aquelas que invocam entrada/saída, então com múltiplos *light-weight processes* que suportem um número igual ou maior de *threads* no nível do usuário é uma abordagem atrativa.

- ... o mapeamento um para um entre *threads* no nível do usuário e *light-weight process* é efetivo para algumas aplicações, por exemplo, um cálculo paralelo de *array* poderia dividir as linhas desse *array* entre diferentes *threads* e se essa relação for um para um nenhum chaveamento de *thread* se fará necessário.

... 2.2.4 - Threads no Windows

- processos suportados por diferentes S.O. diferem em muitos aspectos:
  - como os processos são nomeados, representados e protegidos;
  - como oferecer suporte a *threads*;
  - mecanismos para comunicação entre processos;
  - como processos se relacionam com outros processos.



... 2.2.4 - Threads no Solaris

- processo - processo normal no UNIX que inclui espaço de endereçamento, pilha, e bloco de controle do processo;
- *user-level threads* - implementada através de biblioteca no espaço de endereçamento do usuário, são invisíveis para o sistema operacional;
- *lightweight processes* - podem ser vistos como o mapeamento entre o *user-level threads* e as *threads* no *kernel*;
- *kernel threads* - constituem os elementos fundamentais que podem ser escalonados e direcionados para serem executados em algum dos processadores.

... 2.2.3 - Microkernel - Benefícios

- interfaces uniformes - serviços são providos por meio de envio de mensagens, assim não há distinção para os processos de serviços no nível do *kernel* e do usuário;
- extensibilidade - facilita a adição de novos serviços, bem como o suporte a múltiplos serviços na mesma área funcional (modificações não implicam mudanças no *kernel*);
- flexibilidade - não somente funcionalidades podem ser alteradas mas, também, aspectos já contemplados podem ser retirados e/ou modificados para tornar a implementação mais eficiente;
- portabilidade - aspecto atrativo para sistemas operacionais (todo o código dependente do processador está no *microkernel*);
- confiabilidade - quanto maior o *software*, maior é a dificuldade de se conseguir confiabilidade e nesse sentido, um *microkernel* pode ser rigorosamente testado;

... 2.2.3 - Microkernel - Benefícios

- suporte a sistemas distribuídos - se um sist. distribuído é configurado de tal forma que todos os processos e serviços têm identificação única, então temos como efeito uma única imagem do no nível do *microkernel*:
  - um processo pode enviar uma mensagem sem saber em que máquina o serviço requisitado está residindo e/ou localizado.
- sist. oper. orientados a objetos - a orientação a objetos pode impor disciplina no projeto do *microkernel*, bem como no desenvolvimento de módulos de extensão no sistema operacional:
  - ... como resultado um grande esforço no projeto de *microkernel* tem sido no uso de técnicas de orientação a objetos;
  - ... uma abordagem nessa linha é o uso de componentes (objetos com interfaces bem definidas que podem ser interconectados para formar um *software* na modalidade de *building blocks*).

2.2.3 - Microkernel

- **idéia:** é um pequeno núcleo de sistema operacional que através de um *framework* possibilita futuras extensões (o termo é discutível e tem sido abordado de maneiras diferentes por diferentes sistemas operacionais).

- **algumas questões:**
  - quão pequeno o *kernel* deve ser para ser qualificado como *microkernel* ?
  - como projetar *drivers* de dispositivos para se obter a melhor performance enquanto se abstrai as funções do *hardware*?
  - onde executar operações que não sejam do *kernel* ? ... modo *kernel* ou usuário ?
  - onde manter o código do sub-sistema ?

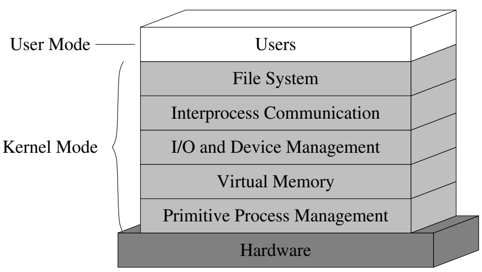
- **histórico:** o seu uso se popularizou no Sist. Oper. Mach.

- ... em teoria esta abordagem provê alto grau de flexibilidade e modularidade — já no Windows NT a modularidade e portabilidade são benefícios chaves.

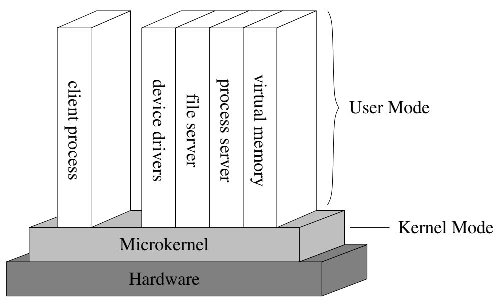
... 2.2.3 - Microkernel

- *layered operating systems* - funções são organizadas hierarquicamente e interações somente se dão entre camadas adjacentes sendo que a maioria das camadas ou suas funcionalidades são executadas no modo *kernel*.

- *microkernel* - somente as funções absolutamente necessárias estão disponibilizadas no *kernel* do sistema operacional, assim, muitas dos serviços e aplicações essenciais são construídos sobre o *kernel* e executados no modo usuário.



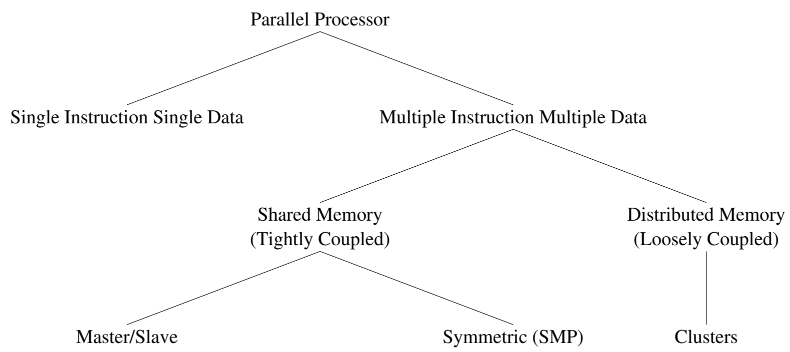
Layered Kernel



Microkernel

2.2.2 - Symmetric Multiprocessing - SMPs

- ... a medida que a tecnologia dos computadores evolui e o custo do *hardware* cai, projetistas têm se deparado com mais e mais situações onde o paralelismo pode ser utilizado — melhorando a performance e até mesmo a confiabilidade;
- ... duas são as abordagens comuns que provêem suporte ao paralelismo: *symmetric multiprocessing* e *clusters* de processadores.
- *symmetric multiprocessing* - ainda bem aceita, a taxonomia introduzida por Flynn enfatiza sistemas de processamento paralelo subdividindo-os em quatro categorias;



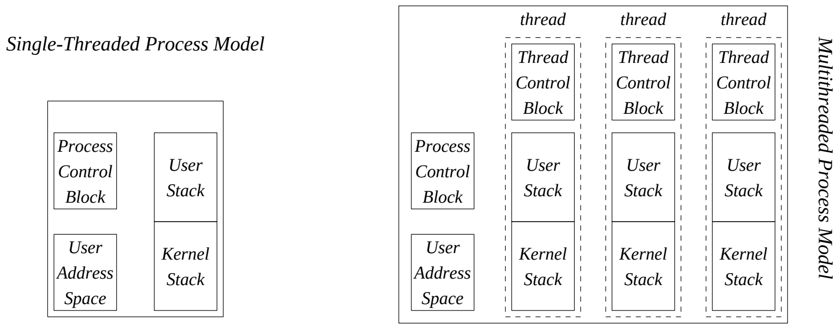
... 2.2.2 - Symmetric Multiprocessing - SMPs

- *clusters* - caracterizados por processadores que possuem memória dedicada e comunicação suportada por rede ou caminhos pré-definidos;
- ... se os processadores compartilham uma memória, então cada processador acessa programas e dados armazenados na memória compartilhada e os processadores comunicam um com outro através da memória — **sistemas multiprocessados com memória compartilhada**.
- ... uma classificação geral de multiprocessadores com memória compartilhada é baseada em como os processos são atribuídos aos processadores:
  - ✯ **master/slave** - o *kernel* do S.O. sempre é executado em um processador em particular tendo os demais a tarefa de executar os processos e utilitários;
  - ✯ **symmetric multiprocessor** - o *kernel* pode ser executado em qualquer processador e, tipicamente, cada processador escalona seu próprio conjunto de processos e *threads* — esta abordagem torna o S.O. mais complexo.

... 2.2.1 - Threads ou Lightweight Process

❖ ... dentro de um processo, podemos ter uma ou mais *threads*, cada uma delas apresentando as seguintes características:

- estado de execução (executando, pronta para ser executada, etc.);
- contexto da *thread* quando esta não estiver sendo executada;
- pilha de execução;
- algumas variáveis locais de armazenamento para cada *thread*;
- acesso à memória e recursos do seu processo, compartilhados com todas as outras *threads* pertencentes ao processo.



... 2.2.1 - Threads ou Lightweight Process

❖ ... a *thread* constitui-se em uma unidade de escalonamento — solução para melhoria do desempenho, por reduzir o tempo para a troca do contexto que, neste caso, é menor que de um processo;

- ... se no entanto, no referirmos ao escalonamento de *threads* pertencentes a processos diferentes, será necessário a troca total de contexto;

- ... um aspecto interessante no conceito de *thread* é a possibilidade de explorarmos a concorrência entre processos através, por exemplo, do compartilhamento de memória.

2.2.1 - Threads ou Lightweight Process

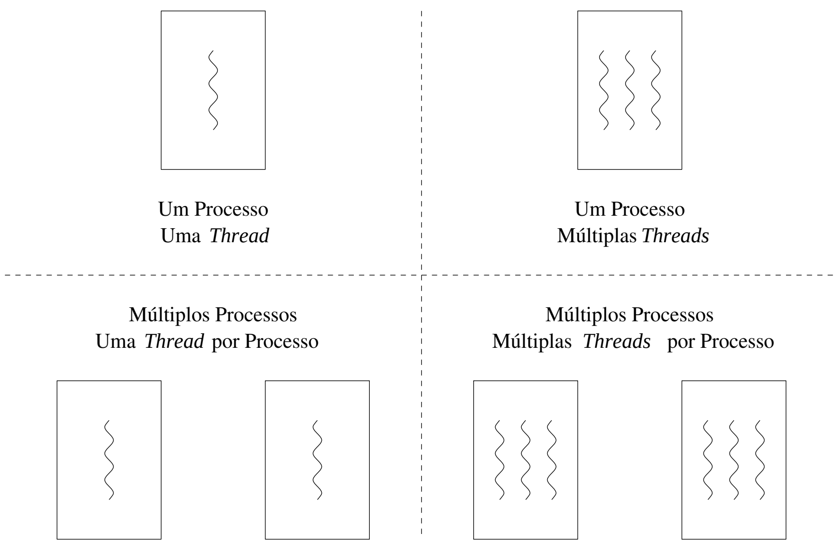
❖ **conceito de processo:** envolve a coordenação (junção) de duas características:

- unidade de alocação de recurso - ... ao processo é atribuído um espaço de endereçamento virtual que de tempo em tempo utiliza-se da memória principal, bem como do controle de outros recursos (p. ex., entrada/saída, arquivos, etc.);
- unidade escalonável: ... define um caminho (rota) de execução em um ou mais programas e, assim possui um estado que a descreve.

- **thread** "*lightweight process*" - surge da idéia de que em alguns S.O. estas duas caracterísiticas são distintas e portanto podem ser tratadas de forma independente;
- *lightweight process* pertencem a um processo e são caracterizadas pelo *program counter*, registradores e pilha;
- *lightweight process* compartilham código, dados e recursos (espaço de endereçamento, variáveis globais, arquivos, etc.) — daí seu nome.
- ... as *threads* operam como processos e, portanto, possuem estado e podem criar outras *threads*, alocar recursos, etc.

... 2.2.1 - Threads ou Lightweight Process

- *multhreading* refere-se a habilidade do sist. oper. suportar múltiplas *threads* dentro de um único processo — algumas abordagens são apresentadas a seguir:



... 2.1.3 - Controle de Processos - Execução do S.O.

- ... como apresentado, uma estrutura típica para imagem de um processo mostra que uma pilha de *kernel* separada é usada para gerenciar as chamadas/retornos enquanto o processo está no modo *kernel*;

- ... códigos e dados do sistema operacional estão em um espaço de endereçamento comum e são compartilhados por todos os processos do usuário;

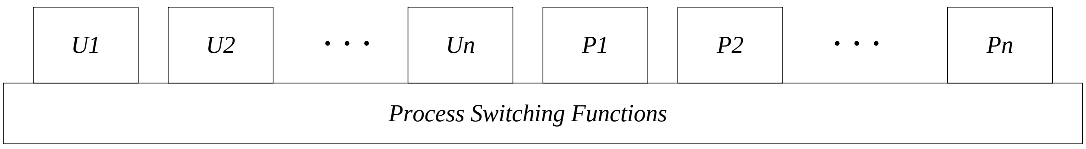
- ... quando uma interrupção, *trap* ou chamada de sistema ocorre, o processador é posto em modo *kernel* e o controle é transferido para o sistema operacional;

- ... isto, no entanto, não significa chaveamento de processo, mas apenas o chaveamento de modo dentro do mesmo processo, ou seja, processo do usuário;

- se, por outro lado, for determinado um chaveamento de processo, então o controle é transferido para a rotina de chaveamento de processos, que pode ou não executar dentro do processo corrente (depende do projeto de sistema);

... 2.1.3 - Controle de Processos - Execução do S.O.

- uma outra alternativa, como ilustrado abaixo, é a visão do sistema operacional como uma coleção de processos do sistema, ou seja, as funções principais do *kernel* estão organizadas como processos separados;



✔ impõe um projeto de programas disciplinado que encoraja o uso de um sistema operacional modular com o mínimo de interfaces claras entre os módulos;

✔ algumas funções de sistema operacional não críticas são convenientemente implementadas como processos separados permitindo a concorrência;

✔ implementar o sistema operacional como um conjunto de processos é útil em ambientes multiprocessados ou em múltiplos computadores, onde alguns serviços do sistema podem ser enviados para processadores dedicados, melhorando a performance.

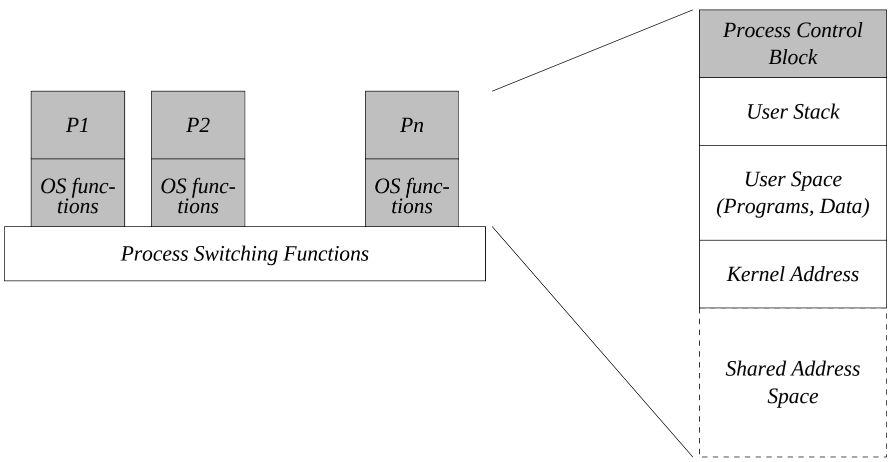
... 2.1.3 - Controle de Processos - Execução do S.O.

- ... o sistema operacional pode executar qualquer função que deseje e depois recuperar o contexto do processo interrompido que a tenha solicitado;
- uma outra alternativa após o sistema operacional armazenar o ambiente do processo é proceder o escalonamento e despacho de um outro processo;
- ... o direcionamento de escalonar um outro processo ou proceder a execução de uma função depende da razão da interrupção e de circunstâncias temporais;
- ... seja qual for o caso, o ponto chave nesta abordagem é que o conceito de processo é aplicável somente a processos do usuário, ou seja, o código do sistema operacional é executado como uma entidade separada que opera em modo privilegiado.

... 2.1.3 - Controle de Processos - Execução do S.O.

✯ funções do sist. oper. executam dentro do processo do usuário - alternativa comum dentro de sistemas operacionais característicos dos mini e microcomputadores.

- ... o sist. oper. é uma coleção de rotinas que o usuário chama para realizar várias funções, sendo que são executadas dentro do ambiente do processo do usuário;



... 2.1.3 - Controle de Processos - Chaveamento de Processos

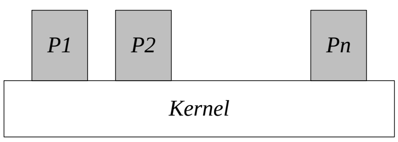
- os passos envolvidos em um chaveamento completo de processos são os seguintes:
  - ❶ salvar o contexto do processo, incluindo contador de programa e registradores;
  - ❷ atualizar o bloco de controle do processo para o processo que está sendo executado, por exemplo, mudar o estado do processo;
  - ❸ mover o bloco de controle do processo para a fila correspondente;
  - ❹ selecionar outro processo para execução segundo algum critério;
  - ❺ atualizar o bloco de controle do processo do processo selecionado;
  - ❻ atualizar a estrutura de dados do gerenciamento de memória;
  - ❼ recuperar o contexto do processador no que se refere ao processo que acaba de ser selecionado, ou seja, carregar os valores prévios para o contador de programa e demais registradores do processo selecionado.

... 2.1.3 - Controle de Processos - Execução do S.O.

- ... alguns aspectos anteriormente apontados no que se refere a definição do sistema operacional como um programa ordinário, podem agora ser melhor esclarecidos !

- Se isto é verdade, é o Sistema Operacional um processo ? Se for, como ele é controlado ? — estas questões tem recebido muitas respostas por parte dos projetistas e a seguir apresentamos algumas abordagens.

✯ *kernel* separado - quando um processo corrente é interrompido ou requeri uma chamada de sistema, o contexto de modo deste processo é salvo e o controle é passado para o *kernel*, que possui sua particular região de memória e de pilha.



... 2.1.3 - Controle de Processos - Chaveamento de Processos

- idéia do chaveamento de processos - em algum instante, um processo executando é interrompido e o sistema operacional atribui outro processo ao estado de executando e muda o controle para aquele processo.

- ... mas como o sist. oper. readquire o controle para efetuar o chaveamento ?!

- Primeiro, o chaveamento de processos pode ocorrer a qualquer instante que o sist. oper. tenha readquirido o controle do processo que está executando correntemente:

  ✰ interrupção - é um sinal externo a execução da instrução corrente;
  ✰ *trap* - associado com a execução da instrução corrente;
  ✰ requisição explícita - chamada de função do sist. operacional.

- ... com uma interrupção, o controle é inicialmente transferido para um manipulador de interrupção, que por sua vez executa algum procedimento básico e depois transfere para a rotina do sist. oper. responsável pela tratamento da interrupção em particular.

... 2.1.3 - Controle de Processos - Chaveamento de Processos

- ... o'utro aspecto a ser investigado é como reconhecer e distinguir entre modo de chaveamento e chaveamento entre processos ?!

- Inicialmente, deve esta claro que o modo de chaveamento (modo usuário ou modo *kernel*) é um modo distinto do chaveamento de processos.

- Segundo, o modo de chaveamento pode ocorrer sem que haja mudança do estado do processo que está sendo executado — neste caso, o *overhead* é baixo.

- ... entretanto, se o processo correntemente executando tem seu estado alterado (*ready*, *blocked*, *ready* e suspenso, etc.), então o sistema operacional deverá efetuar mudanças substanciais no seu ambiente.

... 2.1.3 - Controle de Processos - Modos de Execução

- ... a razão para se utilizar 02 modos é a necessidade de proteger o sistema operacional e suas tabelas chaves, tais quais os blocos de controle de processos.

- ... no modo *kernel*, o *software* tem total controle do processador e de todas as suas intruções, registros e memória — assim este nível de controle não é necessário, por segurança, e não desejável para programas de usuário.

- ?! como o processador sabe em que modo ele está executando e como mudá-lo ?!

  ✯ a primeira questão é respondida com auxílio do registro de *program status word*, que contém um bit indicando o modo;
  ✯ para mudar o modo, a mudança ocorre em resposta a certos eventos, p. ex., quando o usuário invoca um serviço do sistema operacional.

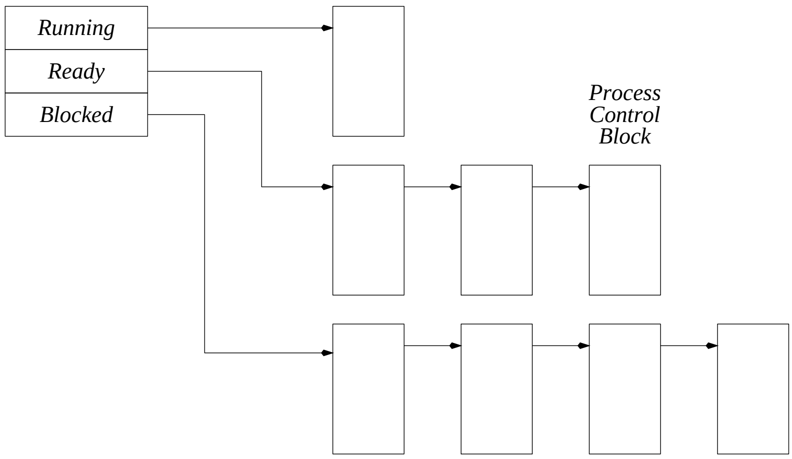
... 2.1.3 - Controle de Processos - Criação de Processos

- Uma vez que o Sistema Operacional decida por criar um processo, seja por que razão for, isto pode ser feito da seguinte maneira:

  ❶ atribuir um único identificador ao novo processo;
  ❷ alocar espaço para o processo (elementos da imagem do processo) - o sist. oper. deve saber quanto espaço para o endereçamento (código e dados) bem como para a pilha (p. ex., estes valores podem ser previamente definidos);
  ❸ inicialização do bloco de controle do processo - isto inclui os campos de identificação do processo, informações do estado do processador e informações de controle do processo;
  ❹ tendo a estrutura de dados sido inicializada, o sist. oper. precisa estabelecer as referências apropriadas entre os vários blocos e tabelas de informações;
  ❺ criação e expansão de outras estruturas de dados a serem utilizadas, por exemplo, na manutenção de um arquivo de contabilidade para cada processo com o propósito de precificação bem como avaliar a sua performance.

... 2.1.2 - Descrição de Processos - Estrutura de Controle

- ... para ilustrar a estrutura de uma lista de processos, considere alguns processos cujos estados estão distribuídos entre *ready*, *blocked* e *running*.



- **Aspecto de Projeto** - rotinas dentro do sistema operacional se farão necessárias para acessar as informações dos blocos de controle dos processos !

2.1.3 - Controle de Processos - Modos de Execução

- ... a maioria dos processos suportam pelo menos 02 modos de execução: modo usuário (modo típico de execução) e modo núcleo (ou com privilégios).
- tipicamente, as funções presentes no *Kernel* do Sistema Operacional, são: gerenciamento de processos, memória, entrada/saída e funções de suporte.

★ Gerenciamento de Processos - criação e finalização; escalonamento e expedição; chaveamento entre processos; sincronização e suporte a comunicação entre processos e gerenciamento dos blocos de controle dos processos;

★ Gerenciamento de Memória - alocação de espaço de endereçamento para processos; *swapping* e gerenciamento de páginas e segmentos;

★ Gerenciamento de Entrada/Saída - gerenciamento de *buffer* e alocação de canais e dispositivos de entrada/saída para processos;

★ Funções de Suporte - manipulação de interrupções; contabilidade e monitoramento.

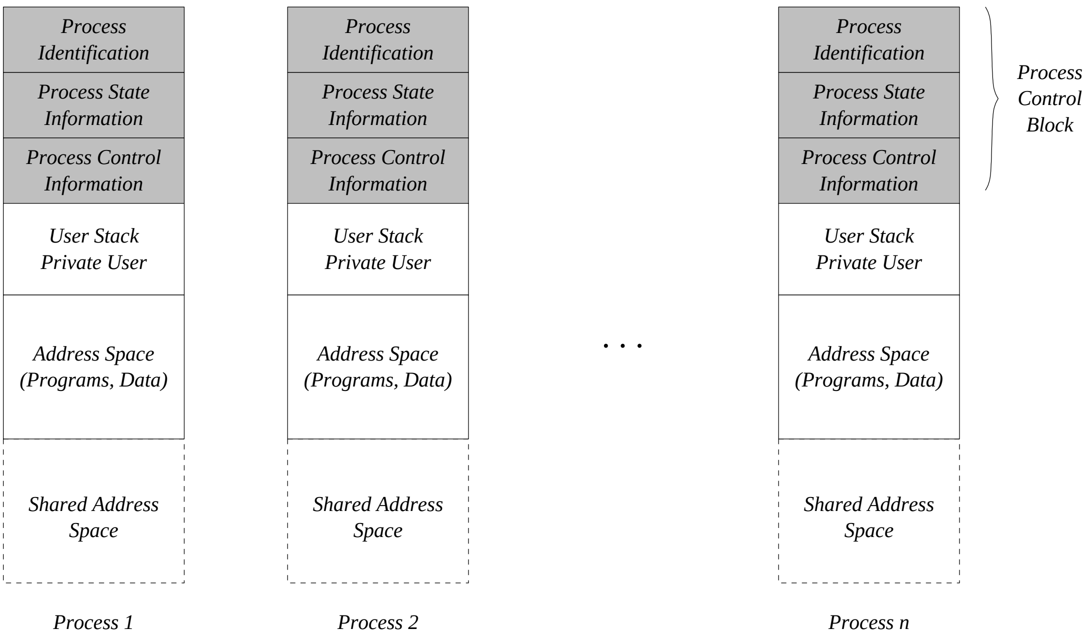
... 2.1.2 - Descrição de Processos - Estrutura de Controle

- ... neste conjunto destacamos:
  - ✯ gerenciamento de memória - esta sessão inclui ponteiros tabelas de segmentos e/ou páginas que descrevem a memória virtual atribuída ao processo;
  - ✯ utilização e propriedade de recursos - recursos controlados por um processo podem ser indicados, p. ex., arquivos abertos bem como a história de utilização do processador e de outros recursos.

- Portanto, o bloco de controle de um processo contém todas as informações necessárias sobre o processo para que o sistema operacional possa gerenciá-lo e controlá-lo.

... 2.1.2 - Descrição de Processos - Estrutura de Controle

- ... a figura abaixo sugere a estrutura da imagem de um processo na memória virtual:



... 2.1.2 - Descrição de Processos - Estrutura de Controle

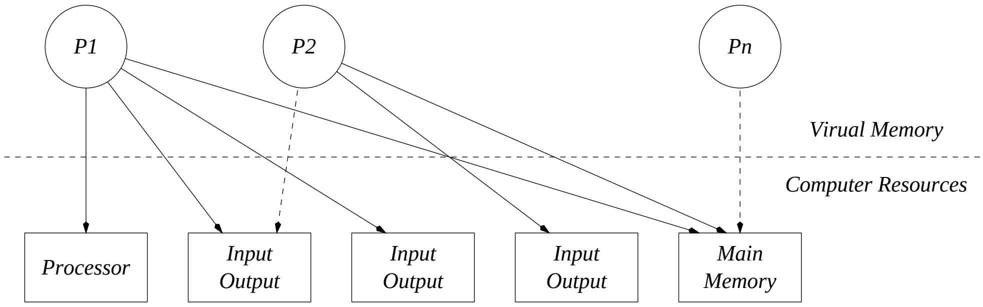
- ... para gerenciar e controlar um processo, o sistema operacional precisa conhecer: onde está localizado o processo e os seus atributos (*process control block*).
- um processo deve no mínimo incluir um programa(s) para ser executado, e pode estar associado a um conjunto de dados, englobar uma pilha e ter atributos;
- ... esta coleção é comumente referenciada como imagem de um processo.
- Em relação aos atributos, podemos agrupar as informações do bloco de controle de processos em três categorias gerais: identificação do processo, informação do estado do processador e informação de controle do processo.
- **Identificação do Processo** - praticamente em todos os sistemas operacionais um número de identificação única é atribuído a cada processo.
  - ... dentre os identificadores estão: identificador do processo, identificador do processo que o criou (processo pai) e identificador do usuário.

... 2.1.2 - Descrição de Processos - Estrutura de Controle

- **Informação do Estado do Processador** - consiste dos conteúdos dos registradores visíveis para o usuário, registradores de controle (p. ex., contador de programa), registrador *program status word* e apontadores de pilha.
- **Informações de Controle do Processo** - esta é a informação adicional necessária para que o sistema operacional controle e coordene os vários processos;
- ... neste conjunto destacamos:
  - ✯ informações de estado e escalonamento - p. ex., estado do processo, prioridade, informações relativas ao escalonamento e eventos;
  - ✯ estrutura de dados - p. ex, fila de processos;
  - ✯ comunicação entre processos - vários *flags*, sinais e mensagens podem estar associadas à comunicação entre processos;
  - ✯ privilégios do processo - processos podem ter privilégios em relação a memória acessada e aos tipos de instruções que podem ser executados;

2.1.2 - Descrição de Processos

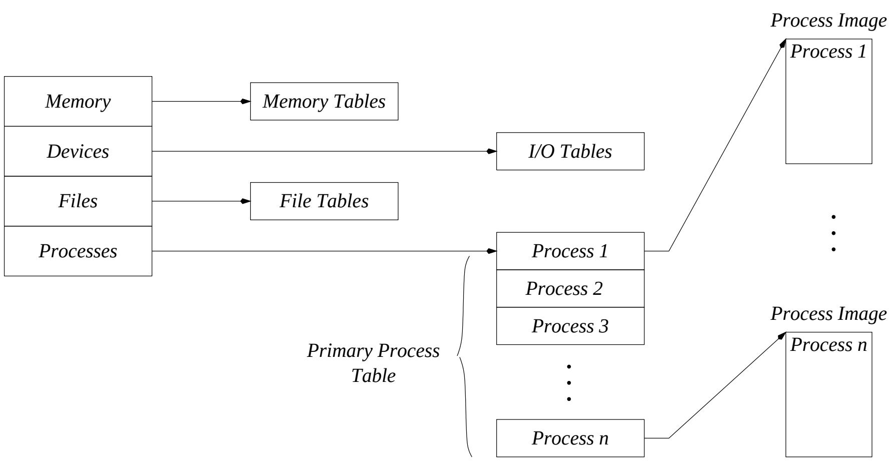
- ... fundamentalmente nós podemos conceber o sistema operacional como uma entidade que gerencia o uso de recursos do sistema através dos processos;
- ... este conceito é ilustrado na figura abaixo, que descreve um ambiente multiprogramado com um número de Processos (P1, P2, ... Pn) que foram criados e que ocupam parte da memória principal.



- ... processos P1 e P2 estão na memória principal, embora P2 esteja bloqueado esperando por um dispositivo de entrada/saída alocado por P1. P3 está suspenso.

... 2.1.2 - Descrição de Processos - Estrutura de Controle

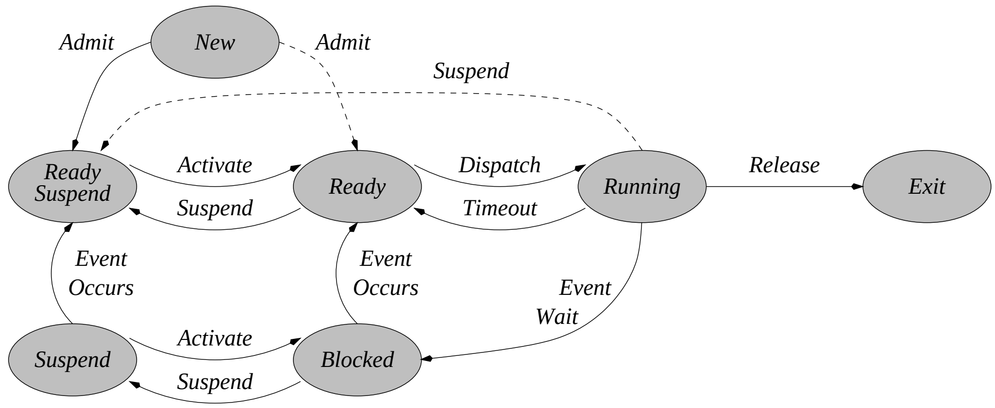
- ... para o sistema operacional gerenciar processos e recursos, ele deve ter informações sobre o *status* corrente de cada processo e recurso — para isto, o sist. oper. instancia e mantém tabelas de informação sobre estas entidades.
- ... fundamentalmente, um sistema operacional mantém informações em quatro categorias: memória, entrada/saída, arquivo e processo.



... 2.1.1 - Estados de um Processo - Processo Suspenso

- quando o sistema operacional trazer um processo suspenso para a memória principal, ele estará trazendo um processo bloqueado — como justificar ?

- ... desde modo temos que repensar este aspecto de projeto, pois temos 02 conceitos diferentes: suspenso ou não suspenso e bloqueado ou não bloqueado.



... 2.1.1 - Estados de um Processo - Processo Suspenso

- ... nós podemos generalizar o conceito de processo suspenso, definindo tais processos como aqueles como as seguintes características:

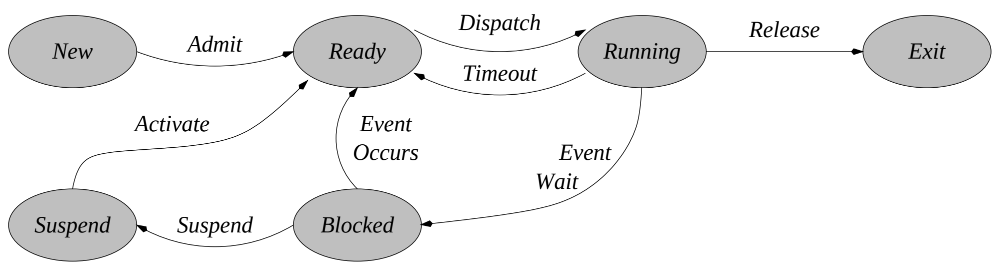
  ❶ o processo não está imediatamente disponível para execução;
  ❷ o processo pode ou não estar esperando por um evento; e caso esteja, sua condição de bloqueado independe da condição de suspenso e em sendo assim, a ocorrência de um evento não habilita o processo a ser executado;
  ❸ o processo foi suspenso por um agente: por ele mesmo, pelo processo pai ou pelo sistema operacional com o propósito de preveni-lo de ser executado;
  ❹ o processo pode não ser removido de seu estado até que o agente ordene explicitamente a remoção desse estado.

- ... algumas razões para suspender um processo:

  ✥ *swapping* - o sist. oper. necessitar liberar espaço na memória principal;
  ✥ temporização - o processo pode ser executado periodicamente e pode ser suspenso enquanto esperando pelo próximo intervalo de execução.

... 2.1.1 - Estados de um Processo - Processo Suspenso

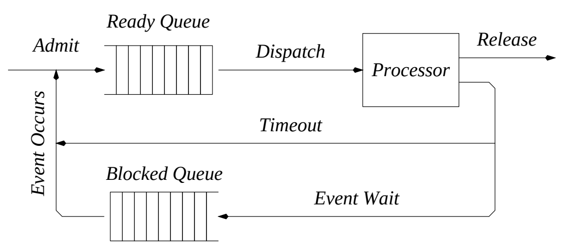
- ... se utilizarmos *swapping* como descrito anteriormente, um novo estado deve ser adicionado ao nosso modelo que descreve o comportamento de um processo:

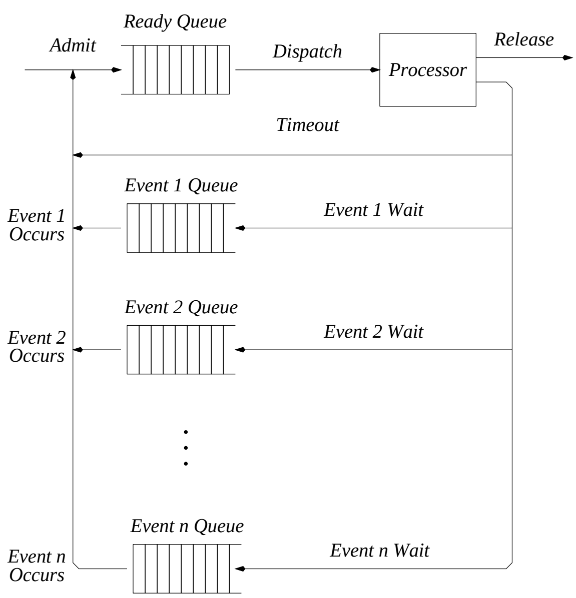


- ... este modelo, no entanto, apresenta alguns problemas:
  - ✘ há duas escolhas para se trazer um processo para o estado de pronto: os sistema pode admitir um processo no estado *new* ou buscar um no estado *suspend*;
  - ✘ quando o sistema operacional trazer um processo suspenso para a memória principal, ele estará trazendo um processo bloqueado — como justificar ?

... 2.1.1 - Estados de um Processo - Processo Suspenso

- ... nas múltiplas filas de processos bloqueados, todos eles em todas as filas precisam estar residentes na memória, assim, como acomodar um novo processo ?





... 2.1.1 - Estados de um Processo - Modelo de 05 Estados

- ... para o exemplo anteriormente apresentado, Processos A, B e C, a tabela abaixo apresenta as transições de cada processo pelos 05 estados:

Time	Processo A	Processo B	Processo C
1-6	**Running**	*Ready*	*Ready*
7-12	*Ready*	*Ready*	*Ready*
13-16	*Ready*	**Running**	*Ready*
17-22	*Ready*	*Blocked*	*Ready*
23-28	*Ready*	*Blocked*	**Running**
29-34	*Ready*	*Blocked*	*Ready*
35-40	**Running**	*Blocked*	*Ready*
41-46	*Ready*	*Blocked*	*Ready*
47-52	*Ready*	*Blocked*	**Running**

... 2.1.1 - Estados de um Processo - Processo Suspenso

- ... os 03 estados anteriormente descritos (*ready, running* e *blocked*) provêem um modo sistemático para modelar o comportamento dos processos, bem como guiar sua implementação em um sistema operacional;

- ... no entanto, há justificativas para acrescentarmos mais estados:
  - cada processo para ser executado deve estar completamente carregado na memória principal e todos os processos na fila devem estar residentes na memória;
  - como acomodar mais processos na memória quando esta está no seu limite ?

- ... algumas possíveis soluções para este problema:
  - expandir a memória principal para acomodar mais processos;
  - mover partes ou todo o processo da memória principal para um memória secundária, ou seja, realizar a operação de *swapping*.

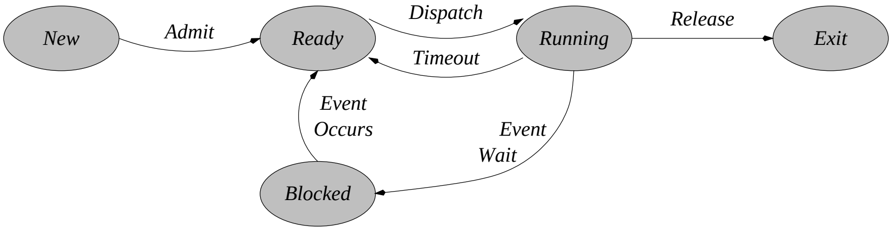
... 2.1.1 - Estados de um Processo - Criação e Finalização

- ... razões possíveis que conduzem a finalização de um processo:
  - falha de entrada/saída - um erro ocorre durante um operação de entrada/saída, p. ex., inabilidade de achar um arquivo;
  - instrução inválida - o processo solicita a execução de uma instrução que não existe, p. ex., um salto para uma área de dados e posterior execução;
  - instrução com privilégio - o processo solicita a execução de uma instrução reservada para o sistema operacional;
  - mal uso de dados - um dado não foi inicializado ou é de um tipo não definido;
  - intervenção do sist. oper. ou operador - p. ex., em um *deadlock*;
  - término do pai - quando o processo pai termina, o sistema operacional pode automaticamente terminar todos os seus descendentes;
  - requisição do pai - o processo pai tipicamente tem a autoridade para terminar qualquer um de seus descendentes.

... 2.1.1 - Estados de um Processo - Modelo de 05 Estados

- como apresentado no modelo de 02 estados, a fila de processos que não estão executando contém tanto processos pronto para serem executados quanto processos bloqueados, o que sugere 02 novos estados: *ready* e *blocked*;

- ... 02 outros estados, *new* e *exit* são adicionados, pois constituem blocos úteis para o gerenciamento de processos:
  - o estado *new* corresponde a um processo que acabou de ser definido;
  - o estado *exit* corresponde ao estado intermediário pelo qual o processo passa antes que a estrutura de dados usada para o seu gerenciamento seja liberaba.



... 2.1.1 - Estados de um Processo - Criação e Finalização

- **criação de um processo** - quando um novo processo é criado, o sistema operacional reserva espaço em memória para estruturas de dados que serão usadas para gerenciar o processo, além de alocar espaço em memória para o processo.

- ... 04 eventos comuns conduzem a criação de um processo, como indicado abaixo:

  ❶ em um sistema *batch*, um processo é criado em resposta à submissão de tarefas;
  ❷ em um ambiente interativo, um processo é criado quando um novo usuário executa o *log in*, ou seja, o usuário abre um sessão de trabalho;
  ❸ um sistema operacional pode criar um processo para executar funções em favor de um programa de usuário, sem que o usuário tenha que esperar (p. ex., impressão);
  ❹ com o propósito de oferecer modularidade ou explorar o paralelismo, um programa de usuário pode inquirir a criação de processos.

- ... quando um processo cria outro, o primeiro é referenciado como um processo pai e o processo criado como processo filho — tipicamente processos com parentesco necessitam comunicar e cooperar uns com os outros.

... 2.1.1 - Estados de um Processo - Criação e Finalização

- ... razões possíveis que conduzem a finalização de um processo:

  ✣ conclusão normal - o processo executa do sistema operacional uma chamada de serviço para informar que ele concluiu sua execução;
  ✣ tempo limite excedido - o processo utilizou o processador por mais tempo que o total especificado, p. ex., *wall clock time*, tempo total executando e etc.;
  ✣ memória indisponível - o processo requer mais memória que o sist. pode prover;
  ✣ violação de limites - o processo tenta acessar um posição de memória cujo acesso não está habilitado para aquele processo;
  ✣ erro de proteção - um processo tenta utilizar um recurso ou um arquivo que não lhe é permitido, p. ex., escrever em um arquivo com permissão apenas para leitura;
  ✣ erro aritmético - o processo solicita um operação proibida, p. ex., divisão por zero, armazenamento de número maior que o permitido;
  ✣ tempo excedido - o processo aguarda um tempo superior ao máximo especificado para que um determinado evento ocorra;

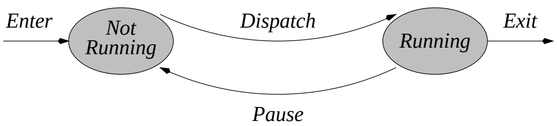
... 2.1.1 - Estados de um Processo

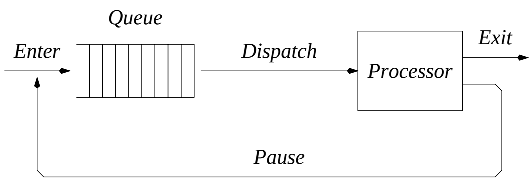
- ... vejamos a execução dos mesmos passos anteriormente apresentados do ponto de vista do processador, considerando que o sistema operacional permite a um processo continuar sua execução por no máximo 06 ciclos de instruções.

1	*alfa + 0*	21	*delta + 4*	41	*delta + 0*
2	*alfa + 1*	22	*delta + 5*	42	*delta + 1*
3	*alfa + 2*	23	*gamma + 0*	43	*delta +2*
4	*alfa + 3*	24	*gamma + 1*	44	*delta + 3*
5	*alfa + 4*	25	*gamma + 2*	45	*delta + 4*
6	*alfa + 5*	26	*gamma + 3*	46	*delta + 5*
	---------- *Time Out*	27	*gamma + 4*	47	*gamma + 6*
7	*delta + 0*	28	*gamma + 5*	48	*gamma + 7*
8	*delta + 1*		---------- *Time Out*	49	*gamma + 8*
9	*delta +2*	29	*delta + 0*	50	*gamma + 9*
10	*delta + 3*	30	*delta + 1*	51	*gamma + 10*
11	*delta + 4*	31	*delta +2*	52	*gamma + 11*
12	*delta + 5*	32	*delta + 3*		---------- *Time Out*
13	*beta + 0*	33	*delta + 4*		
14	*beta + 1*	34	*delta + 5*		
15	*beta + 2*	35	*alfa + 6*		
16	*beta + 3*	36	*alfa + 7*		
	---------- *I/O Operation*	37	*alfa + 8*		
17	*delta + 0*	38	*alfa + 9*		
18	*delta + 1*	39	*alfa + 10*		
19	*delta +2*	40	*alfa + 11*		*delta - Starting Address of*
20	*delta + 3*		---------- *Time Out*		*Program of Process*

... 2.1.1 - Estados de um Processo - Modelo de 02 Estados

- ... a principal responsabilidade do sistema operacional é o controle da execução dos processos, que por sua vez inclui determinar o padrão com o qual os processos são entrelaçados bem como a alocação de recursos requisitados.





- ... o modelo mais simples é aquele que contempla o processo em pelo menos dois estados: processo está sendo executado ou não pelo processador;

- ... assim o processo pode estar em um dos 02 estados: *running* ou *not running*;

- ... processos que não estão sendo executados devem ser mantidos em uma fila, esperando sua vez para serem executados.

2.1 - Descrição e Controle de Processos

- O projeto de um Sistema Operacional deve refletir os requisitos para os quais ele foi projetado, por exemplo, um sistema operacional que suporta multiprogramação deve suportar algumas centenas de usuários;

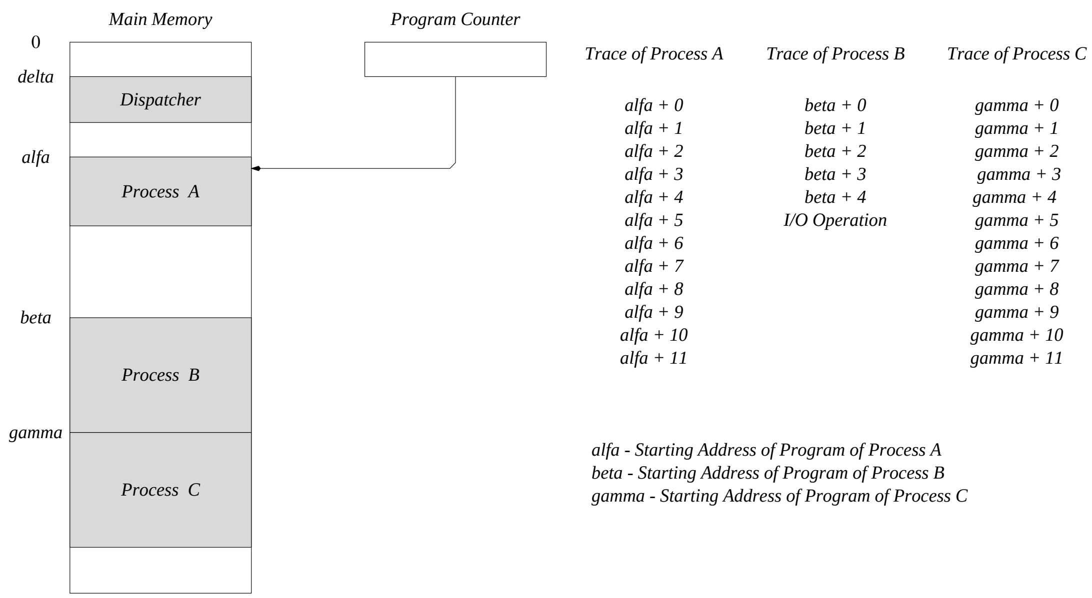
- ... muitos dos requisitos que os sist. oper. precisam suportar podem ser expressos e/ou desenvolvidos em torno do conceito de processos:

  ✯ o sist. oper. deve entrelaçar a execução de vários processos;
  ✯ o sist. oper. deve alocar recursos aos processos segundo políticas estabelecidas;
  ✯ o sistema operacional pode ser requisitado pelo usuário a criar processos ou suportar a comunicação entre processos.

- ... nós iremos começar o estudo detalhado de sistemas operacionais examinando a maneira pela qual eles representam e controlam os processos.

2.1.1 - Estados de um Processo

- ... a função principal de um processador é executar instruções de máquina residentes na memória principal — estas instruções são providas na forma de um programa.



Cap. 2 - Processos

2.1 - Descrição e Controle de Processos

- estados, descrição, controle, ex., UNIX.

2.2 - *Threads*, SMPs e *Microkernel*

- processos e *threads*, multiprocessamento simétrico, *microkernel*, gerenciamento de *threads* e SMPs no Windows e no Solaris (UNIX Sun Microsystems).

2.3 - Concorrência: Exclusão Mútua e Sincronização

- princípios da concorrência, suporte de *hardware* e *software* para exclusão mútua, semáforos, monitores, passagem de mensagem, etc.

2.4 - Concorrência: *Deadlock* e *Starvation*

- *deadlock* - prevenindo, detectando e evitando.
- mecanismos de concorrência no Windows e no UNIX.

Cap. 2 - Processos - Referência Bibliográfica

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

- Cap. 3 - Descrição e Controle de Processos
- Cap. 4 - Threads, SMPs e Microkernel
- Cao, 5 - Concorrência: Exclusão Mútua e Sincronização
- Cap. 6 - Concorrência: Deadlock e Starvation

✯ A.S. TANEMBAUM; "Modern Operating Systems", Prentice-Hall International Editions, 1984, ISBN: 0-13-637331-3

- Cap. 2 - Processos

✯ 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.

- Cap. 2 - Processos