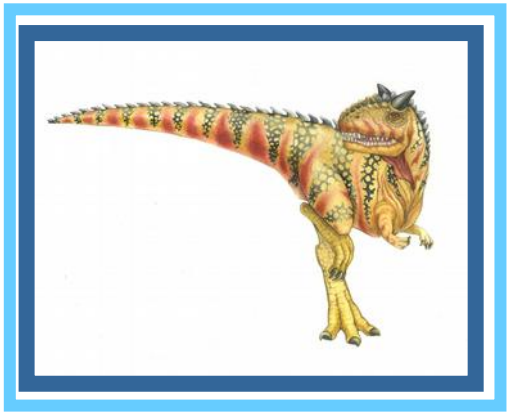
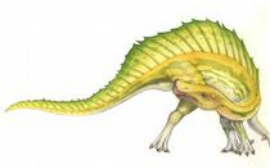
Capítulo 6:
Sincronização de Processos



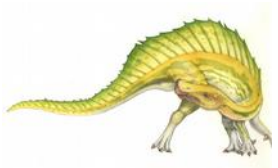
Sumário

- Conceitos relacionados
- O problema da seção crítica
- Hardware de sincronização
- Semáforos
- Problemas clássicos de sincronização
- Regiões críticas
- Monitores
- Exemplos de sincronização em S.O.



Antecedentes

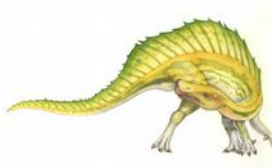
- Acesso concorrente a dados compartilhados pode resultar em inconsistências
  - problemas envolvendo a integridade dos dados compartilhados entre vários processos
- Manter dados consistentes exige mecanismos para garantir a execução cooperativa de processos
- Por exemplo, podemos rever o problema do *buffer* limitado, discutido anteriormente, adicionando um contador de elementos na fila circular:



Buffer limitado (mem.compartilhada)

Dados compartilhados em fila circular

```
#define BUFFER_SIZE 10

typedef struct {
  //...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```



Buffer limitado: produtor

```
item nextProduced;

while (1) {
    //produce an item in nextProduced
    while (counter == BUFFER_SIZE) //buffer cheio
        ; /* do nothing */

    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;

}
```

Buffer limitado: consumidor

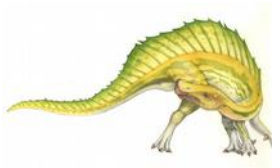
```
item nextConsumed;

while (1) {

    while (counter == 0) // buffer vazio
        ; /* do nothing */

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
    //consume the item in nextConsumed
}
```

Buffer limitado (mem.compartilhada)

- Os comandos abaixo precisam ser executados de forma atômica:

```
counter++; /* produtor */

counter--; /* consumidor */
```

- isto é, não podem ser interrompidos no meio!



Buffer limitado (mem.compartilhada)

Código para `count++` em assembly:

```
a)MOV R1,$counter
b)INC R1
c)MOV $counter,R1
```

Código para `count--` em assembly:

```
x)MOV R2, $counter
y)DEC R2
z)MOV $counter, R2
```

- Cada instrução em linguagem de máquina é independente
- Interrupções podem ocorrer antes/depois de qualquer uma
- Logo, trocas de contexto também podem ocorrer
- As sequências a,b,c e x,y,z podem ocorrer intercaladas

Buffer limitado (mem.compartilhada)

Seja *counter* igual a 5 inicialmente e considere-se a seguinte sequência de execução:

produtor: `MOV R1, $counter` (R1 = 5)

produtor: `INC R1` (R1 = 6)

Troca de contexto

consumidor: `MOV R2, $counter` (R2 = 5)

consumidor: `DEC R2` (R2 = 4)

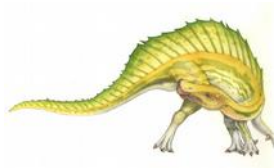
Troca de contexto

produtor: `MOV $counter, R1` (counter = 6)

Troca de contexto

consumidor: `MOV $counter, R2` (counter = 4)

O valor de *counter* pode terminar como 4 ou 6, mas o valor correto seria 5!

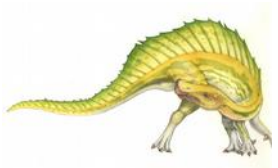


Condição de corrida (*race condition*)

- Situação onde vários processos acessam e manipulam os mesmos dados de forma concorrente
- O resultado final dos acessos aos dados compartilhados depende de quem termina por último!
- Sincronização entre processos pode ser usada para evitar tais corridas
- Pela importância desta questão, boa parte do capítulo é dedicada à *sincronização entre processos* e *coordenação entre processos cooperativos*

O problema de seção crítica

- N processos competem para usar alguma estrutura de dados compartilhada
- Cada processo tem um segmento de código comum onde a estrutura é acessada
- Problema: garantir que quando um processo está executando aquele segmento de código, nenhum outro processo pode fazer o mesmo
  - todo processo executa a uma velocidade não nula
  - não há nenhuma suposição quanto a velocidades relativas dos processos

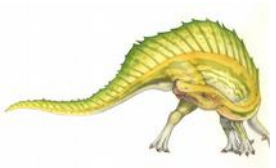


Estrutura geral de um processo Pi

- O problema da seção crítica é projetar um protocolo que os processos possam usar para cooperar
  - cada um deve pedir permissão para entrar na seção crítica

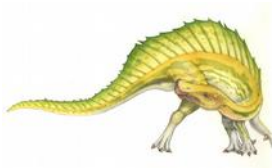
```
do {
    entry section
       critical section
    exit section

    remainder section
} while (1);
```

- Ideia: processos podem compartilhar algumas variáveis para conseguir o controle desejado



O problema de seção crítica

- A solução deve satisfazer os três requisitos abaixo:
  - **Exclusão mútua**: se P*i* está na seção crítica, nenhum outro processo pode entrar nela
  - **Progresso garantido**: se nenhum outro processo está na seção crítica, um processo que tente fazê-lo não pode ser detido indefinidamente
  - **Espera limitada**: se um processo deseja entrar na seção crítica, há um limite na quantidade de vezes que outros processos que podem entrar nela antes dele



Algoritmo 1

- Variável compartilhada: `int turn = 0;`
- Processo i:

```
do {

    while (turn != i) ; //espera a vez do i chegar
       critical section
    turn = j;

    remainder section
} while (1);
```

- Satisfaz exclusão mútua, mas não progresso
  - e se o j não estiver preparado para executar a SC?

Algoritmo 2

- Variável compartilhada: `int flag[2]={0,0};`
- Processo i:

```
do {

    flag[i] = true;
    while (flag[j]) ; //espera o j não estar mais pronto
        critical section
    flag[i] = false;

    remainder section
} while (1);
```

- Satisfaz exclusão mútua, mas não progresso

  e se i e j estiver prontos para entrar na SC e esperarem um pelo outro?

Algoritmo 3 (Peterson)

- Combina ***turn*** e ***flag***
- Processo i:

```
do {

    flag[i] = true; //i está pronto para entrar
    turn = j;       //se o j quiser entrar, pode
    while (flag[j] && turn == j) ;
       critical section
    flag[i] = false;

    remainder section
} while (1);
```

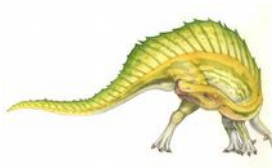
- Satisfaz os três requisitos:
  - *exclusão mútua, progresso garantido, espera limitada*

Hardware de sincronização

- Para simplificar algoritmos, hardware pode prover operação atômica de leitura+escrita

- Exemplo: testa e modifica conteúdo da memória:

```
boolean TestAndSet(boolean* target) {
   boolean old_value = *target;
   *target = true;
   return old_value;
}
```



Exclusão mútua com TestAndSet

- Variável compartilhada:

```
boolean lock=false;
```

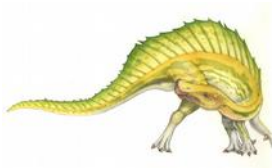
- Processo i:

```
do {

    while ( TestAndSet(&lock) ) /* */;
       critical section
    lock = false;

    remainder section
} while (1);
```

Hardware de sincronização

- Exemplo: troca conteúdo de duas posições

```
void Swap(boolean* a, boolean* b)
{
    boolean tmp = *a;
    *a = *b;
    *b = tmp;
}
```



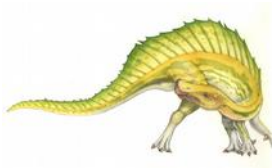
Exclusão mútua com Swap

- Variável compartilhada:

```
boolean lock=false;
```

- Processo i (possui variável local `key`):

```
do {

    key = true;
    while (key) Swap(&lock,&key);
       critical section
    lock = false;

    remainder section
} while (1);
```



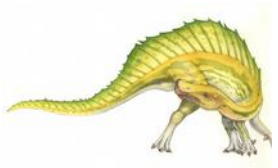
Exclusão mútua por hardware

- Esses algoritmos garantem progresso, mas não espera limitada (por quê?)

  → O problema está no loop de entrada:

  ```
  while ( TestAndSet(&lock) ) /* */;
  ```

  - Ex.: um processo pode sempre ser acordado apenas quando outro já detém o direito de acesso

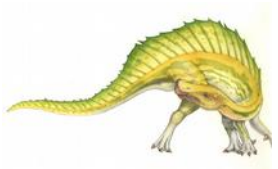


Exclusão mútua “justa”

- Solução: cada processo registra sua intenção de entrar na região crítica separadamente

```
boolean waiting[n]; // = false
```

- Ao sair da região crítica, processo “passa a vez” diretamente ao próximo, se ele existir



Exclusão mútua “justa”

```
do {
   waiting[i] = true;
   while (waiting[i] && TestAndSet(&lock)) ;

      critical section

   j = next(i);
   while ((j!=i) && !waiting[j])     // procura alguém esperando
        j = next(j);

   if (j==i) { lock = false; }       // ninguém esperando
   else      { waiting[j] = false; } // passa a vez diretamente

   waiting[i] = false;

   remainder section
} while (1);
```

Semáforos

- Variável inteira acessível apenas através de duas operações indivisíveis (atômicas):

```
wait(int* s) {
    while (*s<=0) ;
    (*s)--;
}
```

```
signal(int* s) {
    (*s)++;
}
```

- Comumente conhecidas como:
  - P() / V()
  - Down() / Up()

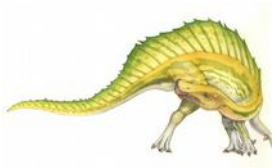
Exclusão mútua com semáforos

- Variável compartilhada:

```
sem mutex = 1;
```

- Processo i:

```
do {
    wait(mutex);
        critical section
    signal(mutex);

    remainder section
} while (1);
```



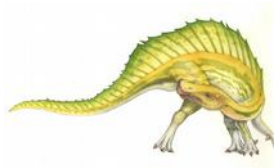
Sincronização com semáforos

- Considere que processo S2 só pode prosseguir depois que processo S1 chegue a um certo ponto.
- Variável compartilhada:

```
sem sync = 0;
```

```
S1:
  // do something
  signal(sync);
```

```
S2:
  wait(sync);
  // do something
```

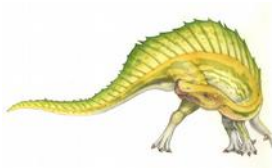


Espera ocupada

- Todas as soluções até agora consomem CPU
  - ex.:

```
while (TestAndSet(&lock)) /**/;
while (*S <= 0) /**/ ;
```

  - semáforo conhecido como *spinlock*
  - exceto nos níveis mais baixos do S.O., isso pode ser evitado
  - outras soluções permitem que processos sejam temporariamente suspensos



Implementação de semáforos

```
typedef struct {
    int value; //valor do semáforo
    struct process *L;  //lista de processos
} sem;
```

```
wait(sem* S) {
  S->value--;
  if (S->value < 0) {
    add(myself,S->L);
    block();
  }
}
```

```
sinal(sem* S) {
  S->value++;
  if (S->value <= 0) {
    P = remove(S->L);
    wakeup(P);
  }
}
```

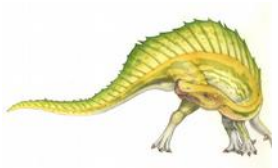
→ OBS.: block() e wakeUp() são chamadas de sistema

Implementação de semáforos

- Forma de gerência da lista de processos bloqueados é extremamente importante
  - **Justiça** (espera limitada): todo processo que executa um wait() deve receber o semáforo em um tempo limitado

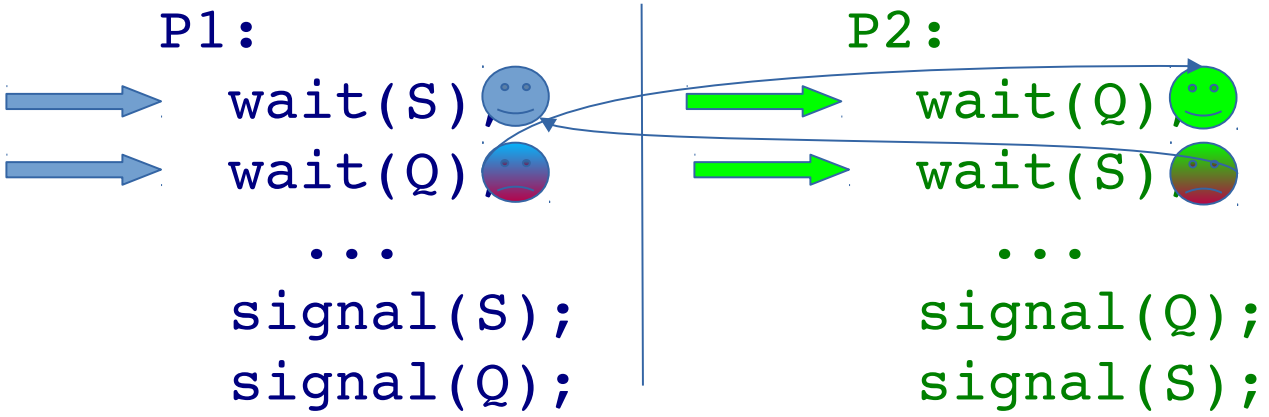
    → solução simples: lista de processos FIFO
  - **Inanição** (*starvation*): conceito importante!

    → o que aconteceria se a lista fosse LIFO?



Deadlocks

- Considere os processos P1 e P2 (S=Q=1):

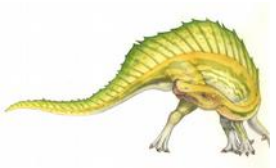


- Que tipo de problema pode ocorrer aqui?
- Problema de aquisição e liberação de recursos
- P1 e P2 ficam impedidos de prosseguir

O capítulo 8 trata de *deadlocks* em detalhes

Semáforos binários

- Ao contrário do anterior (de contagem), só pode assumir os valores 0 e 1
  - diversos *signals* em sequência → apenas um
- Pode ser:
  - mais simples de implementar no HW
  - implementado facilmente por um semáforo de contagem
    - iniciado com 1 (ou zero)
    - todo wait() tem um signal() subsequente



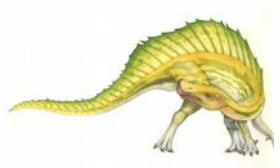
Implementação de semáforos binários

Estrutura de dados para semáforo de contagem:

```
typedef struct {
   int value;
   binSem S1, S2;
} sem;
```

```
wait(sem* S) {
    bin_wait(S->S1);
    S->value--;
    if (S->value < 0) {
        bin_signal(S->S1);
        bin_wait(S->S2);
    }
    bin_signal(S->S1);
}
```

```
sinal(sem* S) {
    bin_wait(S->S1);
    S->value++;
    if (S->value <= 0) {
        bin_signal(S->S2);
    } else {
        bin_signal(S->S1);
    }
}
```



Problemas clássicos de sincronização

- Produtor/consumidor (buffer limitado)
- Leitores e escritores (“gravadores”)
- Jantar dos filósofos (“filósofos famintos”)

→ soluções apresentadas com semáforos

Problema do produtor/consumidor

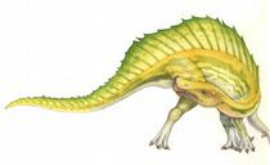
Estruturas de dados:

```
sem mutex      = 1;
sem vacancies  = TAM_BUF;
sem newdata    = 0;
```

```
produtor:
do {
   // produz um item
   wait(vacancies);
      wait(mutex);
      // item → buffer
      signal(mutex);
   signal(newdata);
} while (1);
```

```
consumidor:
do {
   wait(newdata);
      wait(mutex);
      // item ← buffer
      signal(mutex);
   signal(vacancies);
   // processa item
} while (1);
```

OBS.: no livro, vacancies → empty, newdata → full



Leitores e escritores

- Processos compartilham estrutura de dados
- Leitores:
  - fazem apenas leituras (não modificam dados)
  - podem acessar a estrutura concorrentemente
- Escritores:
  - alteram os dados
  - precisam de acesso exclusivo à estrutura

Leitores e escritores

- Ex.: Banco de Dados

- Problema pode ser refinado pelas prioridades:
  - leitores só esperam por escritor em atividade
  - leitores não podem entrar se escritor espera
  - outras combinações possíveis

Leitores e escritores

Estruturas de dados:

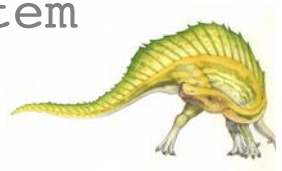
```
sem mutex   = 1;
sem writer  = 1;
int readcnt = 0;
```

//escritor:

```
do {
   // processa algo
   wait(writer);
     // realiza alteração
   signal(writer);
} while (1);
```

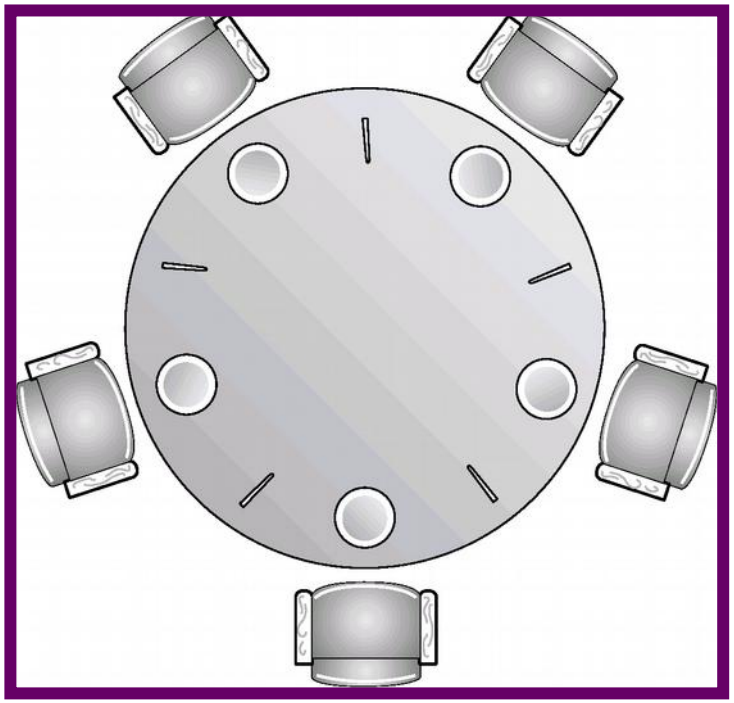
//leitor:

```
do {
   wait(mutex);
     if (readcnt++==0)
       wait(writer);
   signal(mutex);
   // realiza leitura

   wait(mutex);
     readcnt--;
     if (readcnt==0)
       signal(writer);
   signal(mutex);
   // processa item
} while (1);
```



Filósofos famintos

- Filósofos sentam-se ao redor de uma mesa
- É preciso pegar um pauzinho (*hashi* ou *chopstick*) **em cada mão** para comer
- Pegar um utensílio é uma operação atômica
- Problema clássico de alocação de recursos (controle de concorrência)

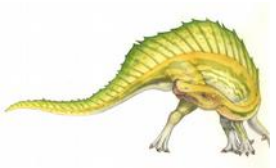


Filósofos famintos

Estruturas de dados:

```
sem chopstick[5] = { 1, 1, 1, 1, 1};
//filósofo i:
do {
        // filosofa
        wait(chopstick[i]);
        wait(chopstick[ (i+1)%5 ]);
        // come
        signal(chopstick[i]);
        signal(chopstick[ (i+1)%5 ]);
} while (1);
```

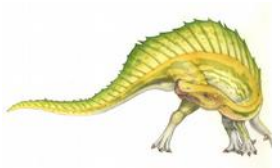
→ que tipo de problema pode ocorrer aqui?



Filósofos famintos: problema

- Garante exclusão mútua, mas...
- E se cada filósofo i:
  - entra em execução e pega o `chopstick[i]`,
  - é interrompido e re-escalonado no fim da fila
  - tenta pegar o `chopstick[ (i+1)%5 ]`
  - mas ele já está tomado!

→ DEADLOCK!



Considerações sobre semáforos

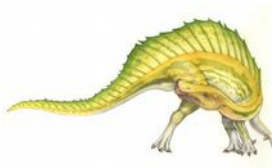
- Semáforos fornecem um mecanismo conveniente e efetivo para sincronização
  - mas.. seu uso incorreto pode resultar em erros difíceis de detectar na execução.

- Lembrando, num semáforo mutex, cada processo deve executar **wait**(mutex) antes de entrar na ***seção crítica*** e **signal**(mutex) depois.
  - se esta sequência não for seguida, dois processos podem entrar em suas seções críticas simultaneamente!
  - essa situação pode ser causado por erro de programação inocente ou por um programador não cooperativo

Considerações sobre semáforos

```
signal(mutex);
    //seção crítica
wait(mutex);
```

- Situação 1:
  - vários processos podem estar executando suas seções críticas simultaneamente
  - violação da exclusão mútua, de difícil detecção em tempo de execução
  - sua descoberta depende de vários processos estarem ativos simultaneamente em suas seções críticas

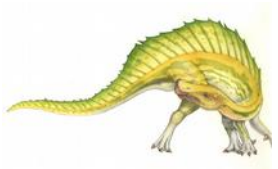


Considerações sobre semáforos

```
wait(mutex);
    //seção crítica
wait(mutex);
```

- Situação 2:
  - um processo obtém o recurso, bloqueando-o e não devolve a mutex, esperando por ela ao invés
  - neste caso um *deadlock* ocorrerá!

- Situação 3: suponha ainda que no programa de um processo, omita-se wait(mutex), ou signal(mutex) ou então as duas operações
  - Violação da exclusão mútua ou *deadlock* ocorrerá!

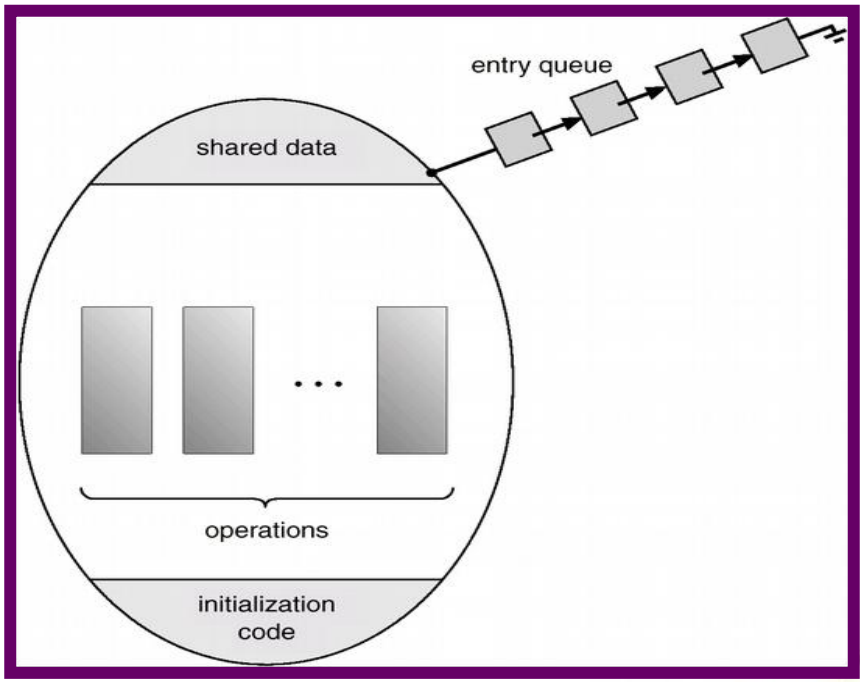


Monitores

- Para lidar com estes problemas decorrentes do uso indevido de semáforos, pesquisadores desenvolveram soluções de linguagem de alto nível, como o construtor do tipo **monitor**

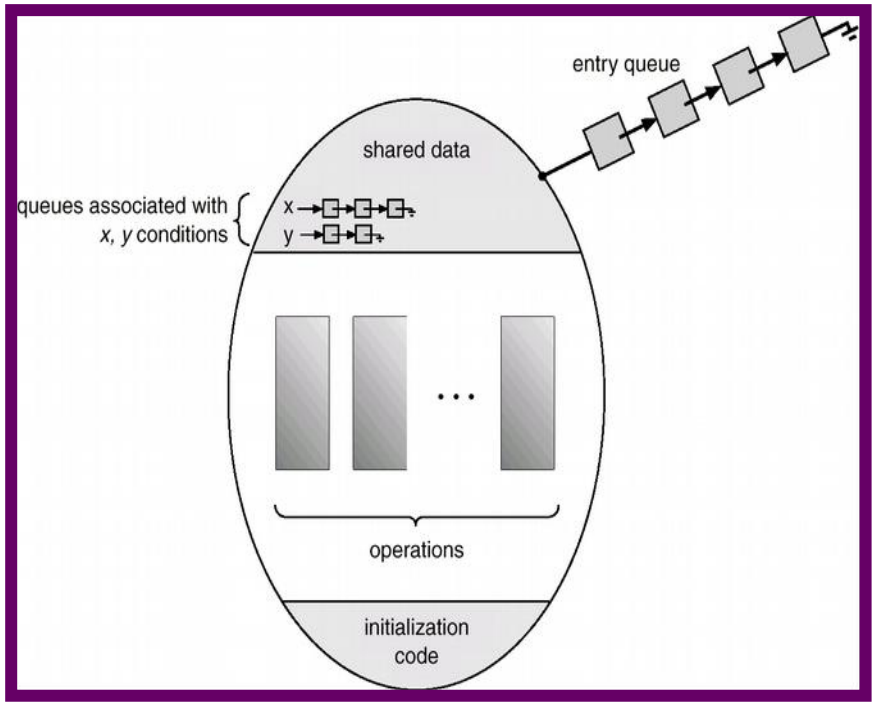
- Um tipo **monitor** é um TAD que apresenta um conjunto de operações que fornecem exclusão mútua dentro do monitor

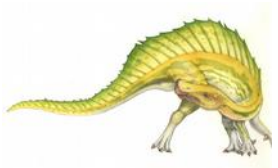
```
monitor nomeDoMonitor{
    //variáveis compartilhadas
    procedure P1(...){...}
    procedure P2(...){...}
    …
    procedure Pn(...){...}
    inicialização(...){...}
}
```



Monitores

- O construtor monitor assegura que **somente um processo de cada vez** fique ativo dentro do monitor.
  - o programador não precisa codificar essa restrição de sincronização explicitamente!
- Porém, tal como definido até o momento, o construtor monitor não é suficientemente poderoso para modelar alguns esquemas de sincronização
  - precisamos definir alguns mecanismos de sincronização adicionais





Monitores personalizados

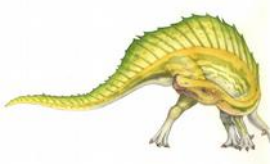
- **Variáveis de condição**: permitem a um procedimento do monitor esperar por uma condição específica
  - `condition x, y; //variáveis x e y`
- Esperar por uma condição específica: `x.wait();`
  - processo espera até ser sinalizado
- Sinalização de uma condição específica: `x.signal();`
  - retoma um único processo suspenso
  - se não há processos esperando, não tem efeito

  → **x** permanecerá como se o *signal* nunca tivesse sido chamado (compare com semáforos :)

Monitor para os filósofos famintos

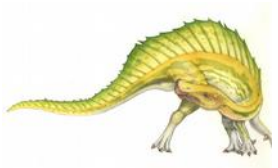
- A solução apresentada a seguir é livre de *deadlocks*!
  - Impõe a restrição de que um filósofo só possa pegar talher se os dois talheres tiverem disponíveis
  - Para tal, é necessário diferenciar três estados em que podemos diferenciar um filósofo: faminto, comendo, pensando

```
monitor FilosofosFamintos {
   enum {THINKING, HUNGRY, EATING} state[5];

   condition self[5]; // variável de condição do filósofo (wait/signal)

   void init() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING; // estado inicial de todos os filósofos
   }

   void pickup(int i);   // invoca ANTES de comer
   void putdown(int i); // invoca APÓS comer
   void test(int i);     // código apresentado adiante
}
```



Monitor para os filósofos famintos

```
void test( int i ) {   // verifica se o filósofo pode comer
    if (
            (state[(i+4)%5] != EATING) // anterior

         && (state[(i+1)%5] != EATING) // próximo

         && (state[i] == HUNGRY)          //filósofo com fome

       ) {   // se os dois talheres estiverem disponíveis
               // e o filósofo estiver faminto, pega-os

           state[i] = EATING;// "pega" os talheres
           self[i].signal;    // sinaliza a condição "peguei"

         } // if
} // test
```



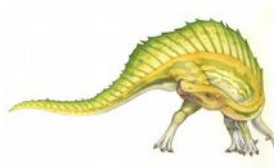
Monitor para os filósofos famintos

```
void pickup( int i ) {
  state[i] = HUNGRY; // o filósofo está com fome

  test(i);           // verifica se pode pegar os talheres

  if (state[i] != EATING) // se não conseguir pegar...
    self[i].wait;         // espera pelos talheres!
}

void putdown( int i ) {
  state[i] = THINKING; // após ter comido, o filósofo pensa e então...

  test( (i+4)%5 );     // tenta passar o talher para o anterior

  test( (i+1)%5 );     // e tenta passar o outro talher para o próximo
}
```

Monitor para os filósofos famintos

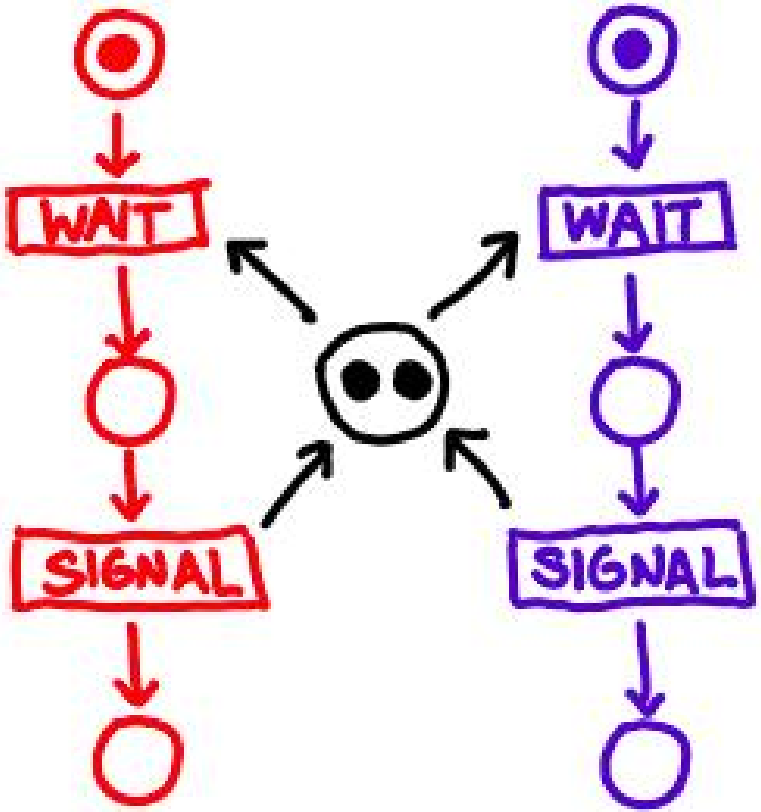
```
// antes de começar a comer, cada filósofo deve pegar os talheres
// o procedimento pickup() pode resultar na suspensão do processo
FilosofosFamintos.pickup(i);

    …
    //come
    …

// depois de comer, o filósofo deve passar adiante os talheres
FilosofosFamintos.putdown(i);
```

- Dois vizinhos poderiam comer simultaneamente? (*Mutex*)
- Seria possível filósofos travarem esperando um ao outro? (*Deadlock*)
- Mas é possível que um filósofo morra de inanição! (*Starvation*)



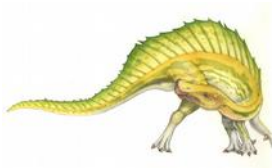
Equivalência entre primitivas

- Qual seria “melhor”? Semáforos ou monitores?
- Todas são funcionalmente equivalentes!
  - É possível implementar um semáforo como um monitor
  - É possível implementar um monitor com semáforos



Transações atômicas

- Conceito originário da área de bancos de dados
- Define um conjunto de operações que deve ser executado atomicamente:
  - ou todas operações executam, ou nenhuma
  - ex.: saque de uma conta bancária
- Operação normalmente baseada em logs
  - registra-se a operação que vai ser realizada
  - executa-se (ou não) a operação
  - o log pode ser consultado no caso de falhas



Fim do Capítulo 6