



# Sistemas Operacionais II

- Sincronização de Processos -

# O Problema das Regiões Críticas

- $N$  processos competindo para utilizar os mesmos dados compartilhados
- Cada processo tem um segmento de código onde é feito o acesso a este dado compartilhado
  - Região crítica
- O problema é garantir que quando um processo executa a sua região crítica, nenhum outro processo pode acessar a sua região crítica
  - Evitar condições de corrida
    - Vários processos acessam dados compartilhados concorrentemente e o resultado da execução depende da ordem específica em que ocorre o acesso ao dado compartilhado

# Região Crítica

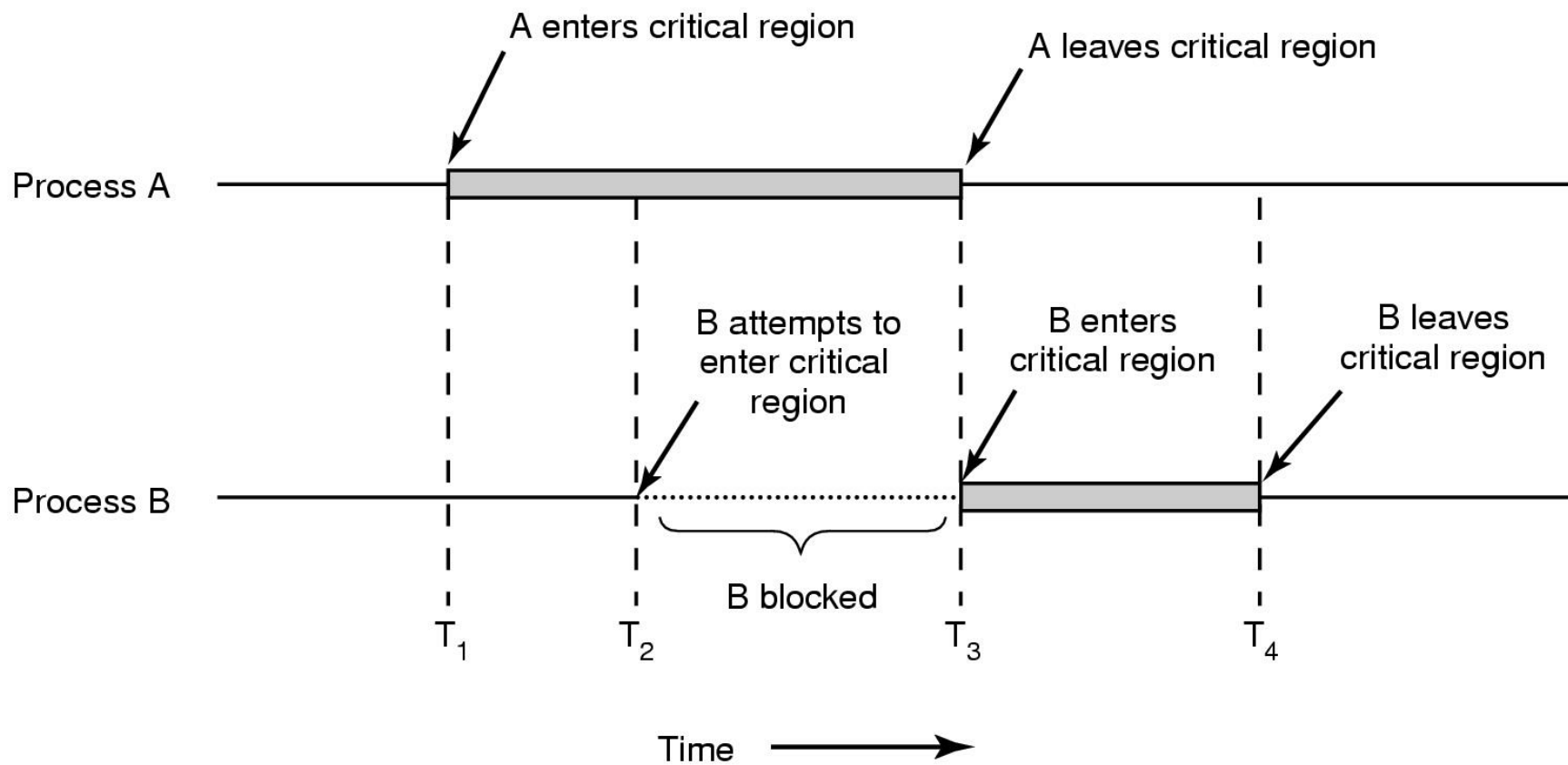
- Dois processos não podem executar em suas regiões críticas ao mesmo tempo
- É necessário um protocolo de cooperação
- Cada processo precisa “solicitar” permissão para entrar em sua região crítica
- Estrutura geral de um processo

```
while (true) {  
    ...  
    Seção de entrada  
    Seção Crítica  
    Seção de Saída  
    ...  
}
```

# Região Crítica

- Para solucionar o problema das regiões críticas alguns requisitos precisam ser satisfeitos:
  - **Exclusão Mútua:** Se um processo  $P_i$  está executando sua região crítica nenhum outro poderá executar a sua região crítica
  - **Progresso:** Nenhum processo fora de sua região crítica pode bloquear outro processo
  - **Espera Limitada:** Um processo não pode esperar indefinidamente para entrar em sua região crítica

# Região Crítica



# Soluções

## ■ Desabilitar Interrupções

**Desabilita Interrupções**

Região Crítica

**Habilita Interrupções**

- Não se deve dar ao processo do usuário o poder de desabilitar interrupções → Se o processo não as reabilita o funcionamento do sistema está comprometido
- As interrupções são desabilitadas em apenas uma CPU
- Exclui não somente processos conflitantes mas também todos os outros processos

# Soluções

- Variável de Bloqueio
  - Não garante a exclusão mútua

```
P  
while (mutex);  
mutex = true;  
Região Crítica;  
mutex = false;
```

$P_1$	$P_2$	mutex
while (mutex) ✗		F
	while (mutex)	F
	mutex = true	T
	RC <sub>2</sub> ✗	T
mutex = true		T
RC <sub>1</sub>		T

# Soluções – Variável de Comutação

- Assegura a exclusão mútua entre dois processos alternando a execução entre as regiões críticas
- A variável *turn* indica qual processo está na vez de executar

```
      PA  
while (turn != A);  
Região Crítica A;  
turn = B;  
Processamento longo
```

```
      PB  
while (turn != B);  
Região Crítica B;  
turn = A;  
Processamento curto
```

- Um processo fora da sua região crítica “bloqueia” a execução do outro



# Soluções – Variável de Comutação

$P_A$	$P_B$	turn
while (turn!=A) ✗		A
	while (turn!=B) ✗	A
$RC_A$ turn = B ✗		A B
	while (turn!=B) $RC_B$ turn = A ✗	B B A
Processamento longo		A

# Soluções – Comutação não Alternada

- Assegura a exclusão mútua entre dois processos sem precisar alternar a execução entre as regiões críticas
- A variável *turn* indica qual processo está na vez de executar
- *Interested* indica se um processo está interessado e pronto para executar sua região crítica
- Um processo entra na sua região crítica se o outro não estiver interessado
- Caso os dois processos estejam interessados o valor de *turn* decide qual processo ganha a região crítica

**P<sub>A</sub>**

```
interested[A] = true;  
turn = B;  
while (interested[B] && turn==B);  
Região Crítica A;  
interested[A] = false;
```

**P<sub>B</sub>**


```
interested[B] = true;  
turn = A;  
while (interested[A] && turn==A);  
Região Crítica B;  
interested[B] = false;
```

# Solução - Instrução *TSL*

- Instruções especiais de hardware que permitem testar e modificar uma palavra de memória atomicamente (sem interrupções)
  - Instrução Test and Set Lock (TSL)

**P**

```
while (lock);  
lock = true;  
Região Crítica;  
lock = false;
```



**P**

```
while TSL(lock);  
Região Crítica;  
lock = false;
```

# Região Crítica

- Todas as soluções apresentadas possuem o problema da espera ocupada
  - O processo “bloqueado” consome tempo de CPU desnecessariamente
- Solução:
  - Introduzir comandos que permitam que um processo seja colocado em estado de espera quando ele não puder acessar a sua região crítica
    - O processo fica em estado de espera até que outro processo o libere

# Semáforos

- Um semáforo é uma variável inteira não negativa que pode ser manipulada por duas instruções P (*Down*) e V (*Up*)
- As modificações feitas no valor do semáforo usando *Down* e *Up* são atômicas
- No caso da exclusão mútua as instruções *Down* e *Up* funcionam como protocolos de entrada e saída das regiões críticas.
  - *Down* é executada quando o processo deseja entrar na região crítica. Decrementa o semáforo de 1
  - *Up* é executada quando o processo sai da sua região crítica. Incrementa o semáforo de 1

# Semáforos

- Um semáforo fica associado a um recurso compartilhado, indicando se ele está sendo usado
- Se o valor do semáforo é maior do que zero, então existe recurso compartilhado disponível
- Se o valor do semáforo é zero, então o recurso está sendo usado

## Down (S)

```
if (S == 0)
    bloqueia processo
else
    S = S - 1;
```

## Up (S)

```
if (tem processo na fila)
    libera processo
else
    S = S + 1;
```

# Semáforos

- Para exclusão mútua é usado um semáforo binário

```
P  
Down (mutex);  
Região Crítica;  
Up (mutex);
```

- Semáforos também são usados para implementar a sincronização entre os processos
- O uso de semáforos exige muito cuidado do programador
  - Os comandos *down* e *up* podem estar espalhados em um programa sendo difícil visualizar o efeito destas operações

# Monitores

- Os monitores são construções de linguagens de programação que fornecem uma funcionalidade equivalente aos semáforos
  - Mais fácil de controlar
- O monitor é um conjunto de procedimentos, variáveis e inicialização definidos dentro de um módulo
- A característica mais importante do monitor é a exclusão mútua automática entre os seus procedimentos
  - Basta codificar as regiões críticas como procedimentos do monitor e o compilador irá garantir a exclusão mútua
  - Desenvolvimento é mais fácil
  - Existem linguagens que não possuem monitores. Os monitores são um conceito de linguagem de programação



# Monitores

**monitor** *monitor-name*

{

declaração de variáveis compartilhadas

**procedure** *P1* (...) {

...

}

**procedure** *P2* (...) {

...

}

**procedure** *Pn* (...) {

...

}

{

código de inicialização

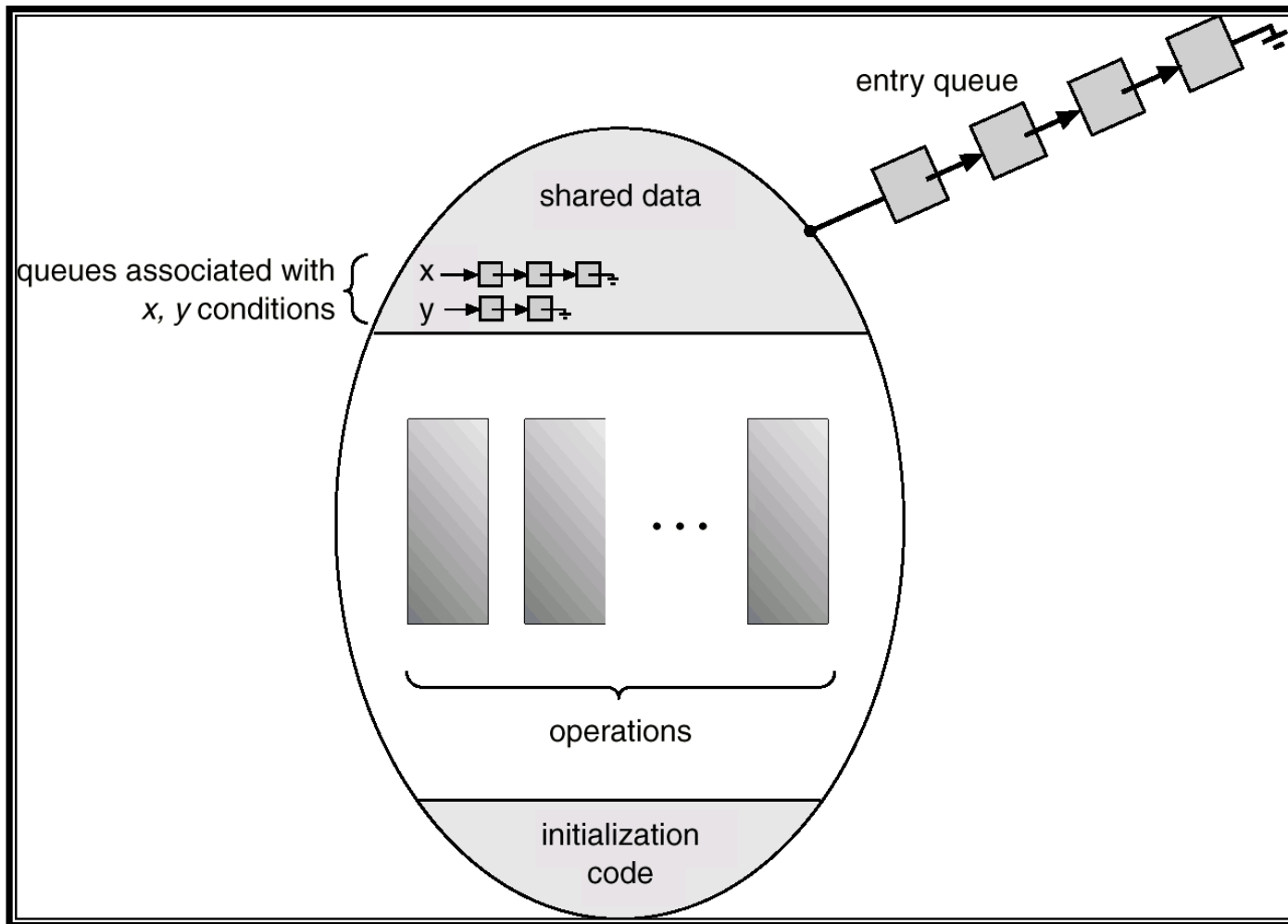
}

}

# Monitores

- Para implementar a sincronização é necessário utilizar variáveis de condição
- Variáveis de condição
  - são tipos de dados especiais dos monitores
  - são operadas por duas instruções *Wait* e *Signal*
- *Wait(C)*: suspende a execução do processo, colocando-o em estado de espera associado a condição C
- *Signal(C)*: permite que um processo bloqueado por *wait(C)* continue a sua execução.  
Se existir mais de um processo bloqueado, apenas um é liberado  
Se não existir nenhum processo bloqueado, não faz nada

# Monitores



# Troca de Mensagens

- Quando é necessário trocar informações entre processos que não compartilham memória
- Usado para comunicação e sincronização
- Basicamente usa duas primitivas
  - *send*(destino, mensagem)
  - *receive*(origem, mensagem)
- Estas duas primitivas podem ser facilmente colocadas em bibliotecas
- Uma biblioteca de comunicação que se tornou padrão é MPI

# Troca de Mensagens

## ■ Sincronização

- Um processo receptor não pode receber uma mensagem até que esta tenha sido enviada
- Deve se determinar o que acontece com um processo após executar um *send* ou *receive*
- *Send* – quando um *send* é executado existe a possibilidade de bloquear ou não o processo até que a mensagem seja recebida no destino
- *Receive* – quando o processo executa um *receive* existem duas possibilidades:
  - se a mensagem já foi enviada o processo a recebe e continua a sua execução
  - se a mensagem ainda não foi enviada:
    - o processo é bloqueado até que a mensagem chegue ou
    - o processo continua a executar e abandona a tentativa de recebimento



# Troca de Mensagens

- *Send* e *Receive* podem ser bloqueantes ou não bloqueantes
  - O mais comum é *send* não bloqueante e *receive* bloqueante
- Endereçamento Direto
  - O processo que envia ou recebe uma mensagem deve especificar a origem e o destino
- Endereçamento Indireto
  - As mensagens não são endereçadas diretamente entre processos origem e destino
  - As mensagens são enviadas para caixas postais (mailboxes)



# Problemas Clássicos de Sincronização

- Produtor/Consumidor
- Jantar dos Filósofos
- Leitores e Escritores
- Barbeiro Dorminhoco

# Produtor/Consumidor

- Um processo produz informações que são gravadas em um buffer limitado
- As informações são consumidas por um processo consumidor
- O produtor pode produzir um item enquanto o consumidor consome outro
- O produtor e o consumidor devem estar sincronizados
  - O produtor não pode escrever no buffer cheio
  - O consumidor não pode consumir informações de um buffer vazio



# Produtor/Consumidor

- Semáforo binário *mutex* para exclusão mútua
- Semáforos *full* e *empty*
  - *Full* conta os espaços cheios no buffer
    - Se *full* igual a zero, então o consumidor deve ser bloqueado
  - *Empty* conta os espaços vazios no buffer
    - Se *empty* igual a zero, então o produtor deve ser bloqueado

# Produtor/Consumidor

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

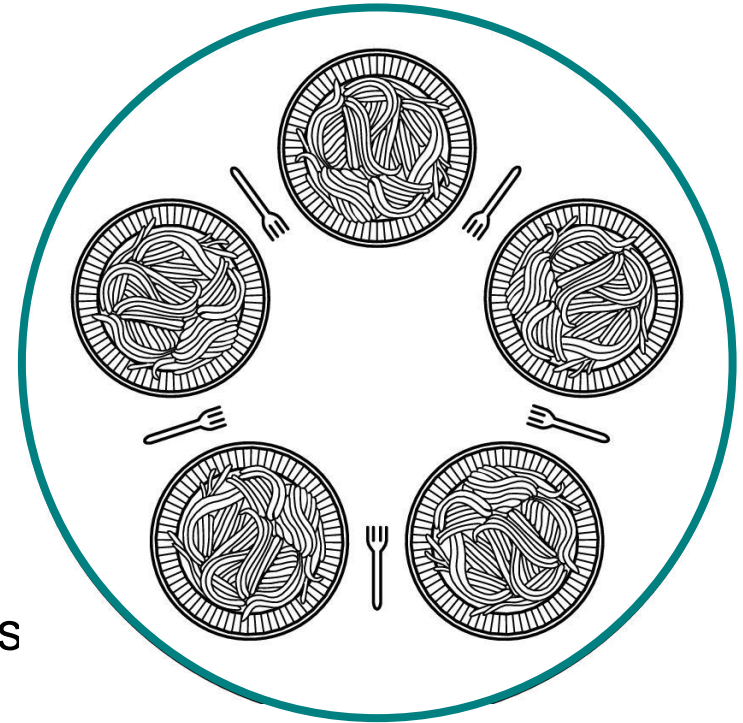
*/\* number of slots in the buffer \*/*  
*/\* semaphores are a special kind of int \*/*  
*/\* controls access to critical region \*/*  
*/\* counts empty buffer slots \*/*  
*/\* counts full buffer slots \*/*

*/\* TRUE is the constant 1 \*/*  
*/\* generate something to put in buffer \*/*  
*/\* decrement empty count \*/*  
*/\* enter critical region \*/*  
*/\* put new item in buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of full slots \*/*

*/\* infinite loop \*/*  
*/\* decrement full count \*/*  
*/\* enter critical region \*/*  
*/\* take item from buffer \*/*  
*/\* leave critical region \*/*  
*/\* increment count of empty slots \*/*  
*/\* do something with the item \*/*


# Jantar dos Filósofos

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



# Jantar dos Filósofos

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

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

- Se todos pegam o garfo da esquerda ao mesmo tempo ocorrerá *deadlock*.

# Jantar dos Filósofos

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

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

# Jantar dos Filósofos

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

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

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

# Leitores e Escritores

- Existem áreas de dados compartilhadas
- Existem processos que apenas lêem dados destas áreas → Leitores
- Existem processos que apenas escrevem dados nestas áreas → Escritores
- Condições:
  - Qualquer número de leitores pode ler o arquivo ao mesmo tempo
  - Apenas um escritor pode acessar o arquivo por vez
  - Se um escritor está escrevendo no arquivo, nenhum leitor poderá utilizá-lo

# Leitores e Escritores

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
```

*/\* use your imagination \*/*  
*/\* controls access to 'rc' \*/*  
*/\* controls access to the database \*/*  
*/\* # of processes reading or wanting to \*/*

*/\* repeat forever \*/*  
*/\* get exclusive access to 'rc' \*/*  
*/\* one reader more now \*/*  
*/\* if this is the first reader ... \*/*  
*/\* release exclusive access to 'rc' \*/*  
*/\* access the data \*/*  
*/\* get exclusive access to 'rc' \*/*  
*/\* one reader fewer now \*/*  
*/\* if this is the last reader ... \*/*  
*/\* release exclusive access to 'rc' \*/*  
*/\* noncritical region \*/*

*/\* repeat forever \*/*  
*/\* noncritical region \*/*  
*/\* get exclusive access \*/*  
*/\* update the data \*/*  
*/\* release exclusive access \*/*



# Barbeiro Dorminhoco

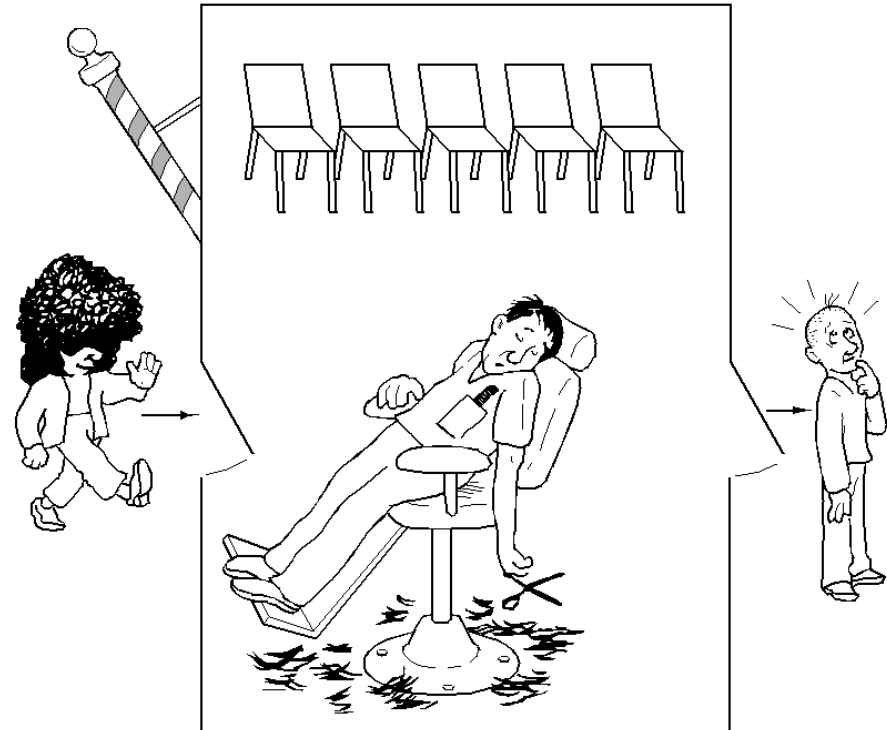
- Neste problema existe:

- 1 barbeiro
- 1 cadeira de barbeiro
- $N$  cadeiras de espera

- Se não houver clientes o barbeiro senta em sua cadeira e dorme

- Quando o cliente chega:

- Ele acorda o barbeiro, caso ele esteja dormindo
- Se o barbeiro estiver trabalhando, o cliente senta para esperar. Caso não existam cadeiras vazias, o cliente vai embora



# Barbeiro Dorminhoco

```
#define CHAIRS 5                                /* # chairs for waiting customers */

typedef int semaphore;                          /* use your imagination */

semaphore customers = 0;                        /* # of customers waiting for service */
semaphore barbers = 0;                         /* # of barbers waiting for customers */
semaphore mutex = 1;                           /* for mutual exclusion */
int waiting = 0;                               /* customers are waiting (not being cut) */

void barber(void)
{
    while (TRUE) {
        down(&customers);                       /* go to sleep if # of customers is 0 */
        down(&mutex);                           /* acquire access to 'waiting' */
        waiting = waiting - 1;                  /* decrement count of waiting customers */
        up(&barbers);                           /* one barber is now ready to cut hair */
        up(&mutex);                             /* release 'waiting' */
        cut_hair();                             /* cut hair (outside critical region) */
    }
}

void customer(void)
{
    down(&mutex);                               /* enter critical region */
    if (waiting < CHAIRS) {                    /* if there are no free chairs, leave */
        waiting = waiting + 1;                 /* increment count of waiting customers */
        up(&customers);                         /* wake up barber if necessary */
        up(&mutex);                             /* release access to 'waiting' */
        down(&barbers);                         /* go to sleep if # of free barbers is 0 */
        get_haircut();                         /* be seated and be serviced */
    } else {
        up(&mutex);                             /* shop is full; do not wait */
    }
}
```