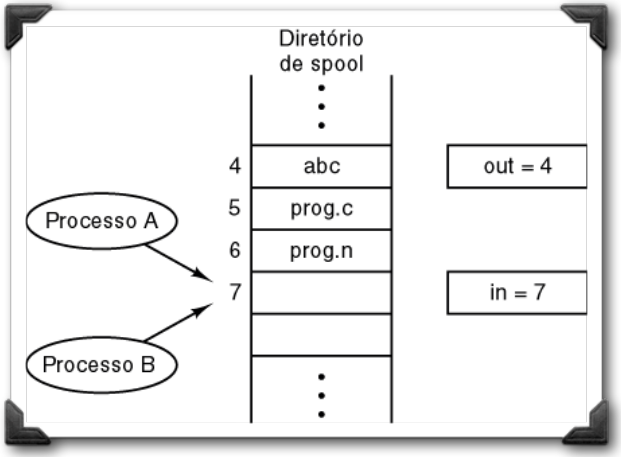
Cap. 2 – Processos
Parte 2

Prof. Marcelo Moreno
moreno@ice.ufjf.br

Comunicação Interprocessos

- Processos/Threads necessitam comunicar-se uns com os outros
  - Colaboração em tarefas
- Comunicação interprocessos
  - Como um processo pode passar informação de um para outro?
  - Como garantir que processos não atrapalhem atividades críticas uns dos outros?
  - Como permitir que sequências de atividades sincronizadas sejam possíveis entre processos?

Condições de Disputa



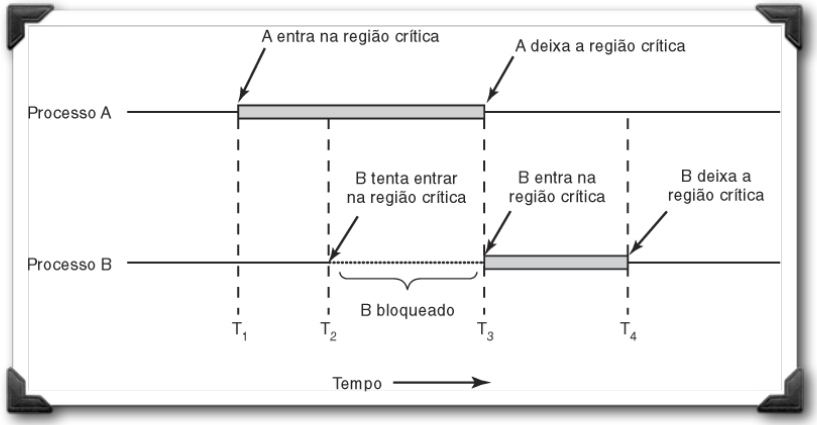
Exclusão Mútua e Regiões Críticas

- Compartilhamento de recursos
  - Espaços de memória compartilhada, arquivos compartilhados
  - Sujeito a condições de corrida e outros problemas
  - Precisamos impedir que mais de um processo leia e escreva o recurso ao mesmo tempo
- Exclusão Mútua
  - Meio que garanta o impedimento de uso de um recurso compartilhado caso ele já esteja em uso
- Região Crítica
  - Parte de um programa em que há acesso a um recurso compartilhado
  - Objetivo: prover mecanismos para impedir que mais de um processo esteja em sua região crítica

Regiões Críticas

- **Quatro condições necessárias para prover exclusão mútua entre regiões críticas:**
  - **Nunca dois processos simultaneamente em uma região crítica**
  - **Nenhuma afirmação sobre velocidades ou número de CPUs**
  - **Nenhum processo executando fora de sua região crítica pode bloquear outros processos**
  - **Nenhum processo deve esperar eternamente para entrar em sua região crítica**

Regiões Críticas



Exlusão Mútua - Desabilitando Interrupções

- **Antes de iniciar região crítica, processo desabilita interrupções**
  - **Exclusão mútua ao extremo**
  - **Nem o escalonador de processos conseguiria a CPU!**
  - **Processo poderia executar sua seção crítica e depois reabilitaria as interrupções**
- **Problemas**
  - **Impossível dar tal liberdade a processos de usuário**
    - E se um programador descuidado se esquece de reabilitar as interrupções após a região crítica em seu programa?
- **Muito usado em pequenos trechos de código do núcleo**

Exclusão Mútua - Variáveis de trava

- **Uma variável em espaço de memória compartilhada serviria de trava ou impedimento**
  - **Variável lock inicia com 0**
  - **Se processo deseja entrar em região crítica, verifica lock**
    - Se lock for 0, altera o valor para 1 e entra na região crítica
    - Se lock for 1, aguarda que ela se torne 0
- **Resolve o problema?**
  - **Não! Condição de disputa!**

Exclusão mútua - Espera Ocupada

- **Variável compartilhada que representa de quem é a vez de entrar em região crítica**
  - **Processo que deseja entrar na região crítica deve verificar continuamente a variável de "vez"**
    - Quando o valor representa sua vez, entra na região crítica
    - Se não, continua verificando em loop, continuamente
  - **Espera ocupada**
  - **Alternância obrigatória**



Espera Ocupada - Alternância Obrigatória

```
while (TRUE) {
    while (turn !=0)          /* laço */ ;
    critical_region( );
    turn = 1;
    noncritical_region( );
}
```

(a)

```
while (TRUE) {
    while (turn !=1)          /* laço */ ;
    critical_region( );
    turn = 0;
    noncritical_region( );
}
```

(b)

- **Resolve o problema?**
  - **Sim, mas não satisfatoriamente**
  - **Regra: "Nenhum processo fora de uma região crítica pode bloquear outro processo"**



Solução de Peterson

- **Não exige alternância**
- **Rotinas para entrada e saída da região crítica**
  - **enter_region**
    - Quem chama enter_region se diz primeiro interessado em entrar em região crítica
    - A vez é dada a ele, e se não houver interesse do outro processo ele é liberado a entrar
    - Caso contrário, continua em espera ocupada
  - **leave_region**
    - Quem chama leave_region simplesmente se diz não-interessado em entrar em região crítica



Solução de Peterson

```
#define FALSE  0
#define TRUE   1
#define N      2                    /* número de processos */

int turn;                           /* de quem é a vez? */
int interested[N];                  /* todos os valores inicialmente em 0 (FALSE) */

void enter_region(int process)      /* processo é 0 ou 1 */
{
    int other;                      /* número de outro processo */

    other = 1 – process;            /* o oposto do processo */
    interested[process] = TRUE;     /* mostra que você está interessado */
    turn = process;                 /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE)  /* comando nulo */;
}

void leave_region(int process)      /* processo: quem está saindo */
{
    interested[process] = FALSE;    /* indica a saída da região crítica */
}
```



Espera Ocupada - Instrução TSL

- **Test and Set Lock**
  - **Testar e atualizar a variável de trava**
  - **TSL RX, LOCK**
    - Lê o conteúdo da palavra de memória LOCK no registrador RX
    - Armazena um valor diferente de zero (0) no endereço de lock
    - Operações em uma sequência indivisível
  - **Podem ser escritas rotinas em código de montagem usando tal instrução, para entrada e saída em regiões críticas**

Espera Ocupada - Instrução TSL

```
enter_region:
    TSL REGISTER,LOCK          | copia lock para o registrador e põe lock em 1
    CMP REGISTER,#0            | lock valia zero?
    JNE enter_region           | se fosse diferente de zero, lock estaria ligado,
                                 portanto continue no laço de repetição
    RET | retorna a quem chamou; entrou na região crítica

leave_region:
    MOVE LOCK,#0               | coloque 0 em lock
    RET | retorna a quem chamou
```

Produtor/Consumidor

- **A.k.a. Problema do buffer limitado**
  - **2 Processos compartilham um buffer de tamanho fixo**
    - Processo Produtor: Armazena informação no buffer
    - Processo Consumidor: Retira informação do buffer
  - **O Problema:**
    - Buffer pode estar cheio e Produtor tem informação a armazenar
    - Buffer pode estar vazio e Consumidor quer retirar informação
  - **Solução: Colocar processos para dormir quando impossibilitados de operar o buffer e depois acordá-los quando for possível**

Dormir/Acordar

- **Primitivas de comunicação entre processos que bloqueiam**
  - **Não gastam tempo de CPU durante o bloqueio**
  - **A chamada Sleep**
    - Faz com que o processo chamador seja suspenso
    - Até que outro processo o desperte novamente
  - **A chamada Wakeup**
    - Faz com que o processo informado como parâmetro seja despertado

Dormir/Acordar

```
#define N 100                                   /* número de lugares no buffer */
int count = 0;                                  /* número de itens no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* número de itens no buffer */
          item = produce_item( );               /* gera o próximo item */
          if (count == N) sleep( );             /* se o buffer estiver cheio, vá dormir */
          insert_item(item);                    /* ponha um item no buffer */
          count = count + 1;                    /* incremente o contador de itens no buffer */
          if (count == 1) wakeup(consumer);     /* o buffer estava vazio? */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          if (count == 0) sleep( );             /* se o buffer estiver vazio, vá dormir */
          item = remove_item( );                /* retire o item do buffer */
          count = count – 1;                    /* decresça de um o contador de itens no buffer */
          if (count == N – 1) wakeup(producer); /* o buffer estava cheio? */
          consume_item(item);                   /* imprima o item */
     }
}
```

Não funciona!



Semáforos

- **Dijkstra (1965) sugere o uso de variável Semáforo**
  - **Variável inteira compartilhada para contar número de sinais de acordar**
    - Valor 0: Nenhum sinal de acordar foi salvo
    - Valor positivo: Um ou mais sinais de acordar estão pendentes
  - **Semáforos são manipulados por duas operações**
    - Down: Verifica se o valor do semáforo tem valor:
      - Se >0, decrementa o semáforo - gasta um sinal de wakeup
      - Se 0, o processo é posto para dormir, sem terminar down
      - Da verificação ao adormecer, uma ação atômica
    - Up: Incrementa o valor de um semáforo
      - Se há processos dormindo naquele semáforo, um deles é acordado
      - Incremento e sinalização são uma ação atômica



Semáforos

```
semaphore fillCount = 0;
   semaphore emptyCount = BUFFER_SIZE;

   procedure producer() {
        while (true) {
             item = produceItem();
             down(emptyCount);
                  putItemIntoBuffer(ite
             up(fillCount);
        }
   }

   procedure consumer() {
        while (true) {
             down(fillCount);
                  item =
   removeItemFromBuffer();
             up(emptyCount);
              consumeItem(item);
        }
```

Somente 1 Produtor e 1 Consumidor!



Semáforos

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0 ;                            /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE é a constante 1 */
          item = produce_item( );               /* gera algo para pôr no buffer */
          down(&empty);                         /* decresce o contador empty */
          down(&mutex);                         /* entra na região crítica */
          insert_item(item);                    /* põe novo item no buffer */
          up(&mutex);                           /* sai da região crítica */
          up(&full);                            /* incrementa o contador de lugares preenchidos */
     }
}

void consumer(void)
{
     int item;

     while (TRUE) {                             /* laço infinito */
          down(&full);                          /* decresce o contador full */
          down(&mutex);                         /* entra na região crítica */
          item = remove_item( );                /* pega o item do buffer */
          up(&mutex);                           /* deixa a região crítica */
          up(&empty);                           /* incrementa o contador de lugares vazios */
          consume_item(item);                   /* faz algo com o item */
     }
}
```


Mutexes

- **Mutexes são semáforos simplificados**
  - **Usados quando não precisamos da capacidade de contar dos semáforos**
  - **Adequado apenas para a exclusão mútua**
  - **Possui dois estados: Impedido e Desimpedido**
- **Possui duas operações**
  - **mutex_lock: Para entrada na região crítica**
    - Se o mutex estiver desimpedido, processo pode continuar
    - Caso contrario, processo será bloqueado
  - **mutex_unlock: Saída da região crítica**
    - Acorda um dos processos bloqueados naquele mutex



Mutexes

```
mutex_lock:
     TSL REGISTER,MUTEX          | copia mutex para o registrador e o põe em 1
     CMP REGISTER,#0             | o mutex era zero?
     JZE ok                      | se era zero, o mutex estava desimpedido, portanto retorne
     CALL thread_yield           | o mutex está ocupado; escalone um outro thread
     JMP mutex_lock              | tente novamente mais tarde
ok:  RET | retorna a quem chamou; entrou na região crítica


mutex_unlock:
     MOVE MUTEX,#0               | põe 0 em mutex
     RET | retorna a quem chamou
```



Semáforos e Mutexes

- **Excelentes mecanismos de comunicação entre processos**
  - **Semáforos vão além da exclusão mútua, permitindo sincronização mais rica entre processos**
- **Fáceis de usar, se o programador for cuidadoso**
- **Erros podem ocorrer...**
  - **Inverter a ordem de sinalização de um semáforo de exclusão mútua com o de sincronização**
  - **Deadlock!**



Monitores

- **1974/75. Proposta de uma unidade básica de sincronização de processos de alto nível**
  - **Monitor: Coleção de procedimentos, variáveis e estruturas de dados agrupada em um módulo**
  - **Processos chamam procedimentos de um monitor quando necessário, mas não têm acesso a estruturas internas**
- **Propriedade relevante: Somente um procedimento pode estar ativo em um monitor em um dado momento**
  - **Construto oferecido pela linguagem de programação**



Monitores

- **Quando um processo chama um procedimento do monitor:**
  - **Rotina verifica se qualquer outro processo já está ativo no monitor**
    - Caso exista um processo ativo no monitor, o processo que chamou o procedimento é suspenso
    - Caso contrário, é permitida a entrada no monitor e o procedimento é executado
  - **É o compilador o responsável por gerar código que implemente a exclusão mútua**
    - Menor propensão a erros
- **Cada região crítica pode ser, portanto, um procedimento do monitor**



Monitores

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .
        .
        .
        end;
end monitor;
```



Monitores

- **Exclusão mútua: OK!**
- **Sincronização?**
  - **Variáveis condicionais, manipuladas por operações wait e signal.**
    - Mas, lembrando, não pode-se permitir que dois processos estejam ativos no monitor ao mesmo tempo.
    - Alguém deve deixar o monitor depois de um signal
    - Hoare: processo acordado suspende o outro
    - Hansen: signal somente usado ao fim de um procedimento
    - Ou, ainda: processo que fez signal vai até o fim do procedimento, deixando o monitor



Monitores

- **Variáveis condicionais não são contadores!**
- **Mas por que não há o problema ocorrido com wakeup/sleep?**
  - **Lembrando, havia condição de corrida, pois wakeup/sleep não oferecem exclusão mútua**
  - **Variável condicional é usada dentro de monitores, provendo exclusão mútua em seu uso**
  - **Não há como perder wakeups indevidamente**



Monitores

```
monitor ProducerConsumer
    condition full, empty;
    integer count;
    procedure insert(item: integer);
    begin
        if count = N then wait(full);
        insert_item(item);
        count := count + 1;
        if count = 1 then signal(empty)
    end;
    function remove: integer;
    begin
        if count = 0 then wait(empty);
        remove = remove_item;
        count := count – 1;
        if count = N – 1 then signal(full)
    end;
    count := 0;
end monitor;
```

```
procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end;
procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end;
```



Monitores em Java

```
static class our_monitor {                  // este é o monitor
    private int buffer[ ] = new int[N];
    private int count = 0, lo = 0, hi = 0; // contadores e índices
    public synchronized void insert(int val) {
        if (count == N) go_to_sleep( );     // se o buffer estiver cheio, vá dormir
        buffer [hi] = val;                  // insere um item no buffer
        hi = (hi + 1) % N;                  // lugar para colocar o próximo item
        count = count + 1;                  // mais um item no buffer agora
        if (count == 1) notify( );          // se o consumidor estava dormindo, acorde-o
    }
    public synchronized int remove( ) {
        int val;
        if (count == 0) go_to_sleep( );     // se o buffer estiver vazio, vá dormir
        val = buffer [lo];                  // busca um item no buffer
        lo = (lo + 1) % N;                  // lugar de onde buscar o próximo item
        count = count – 1;                  // um item a menos no buffer
        if (count == N – 1) notify( );      // se o produtor estava dormindo, acorde-o
        return val;
    }
    private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
}
```



Monitores e Semáforos

- **Semáforos são muito fáceis de serem implementados e usados**
  - **Exigem cuidado do programador na especificação de exclusão mútua e sincronização**
- **Monitores possuem nível de abstração ainda mais alto, fáceis de usar e menos suscetíveis a erros**
  - **Exigem que a linguagem de programação ofereça tal mecanismo**
- **Ambos são ótimos para IPC em máquinas multiprocessadas**
  - **Implementação por memória compartilhada**
  - **TSL**
- **Mas e se os processadores estão distribuídos em uma rede?**



Troca de Mensagens

- **Duas primitivas:**
  - **Send(destino, &msg);**
    - Envia uma mensagem para um certo destino
  - **Receive(origem, &msg);**
    - Recebe uma mensagem de uma certa origem
    - Se nenhuma mensagem estiver disponível, o receptor poderá ficar bloqueado até que uma mensagem chegue
      - Alternativamente, pode-se retornar um erro imediatamente
- **Redes de comunicação são tão confiáveis quanto memória compartilhada?**
  - **Mensagem de reconhecimento (ACK)!!!**



Troca de Mensagens

- Se reconhecimento não for devolvido depois de certo tempo
  - Fazer retransmissão da mensagem!!
- Mas e se a mensagem foi corretamente recebida e o que se perdeu foi o reconhecimento?
  - Duplicou-se a mensagem!!!
  - Fazer sequenciação!!!
- Endereçamento
- Partiremos do pressuposto que mensagens são armazenadas (enfileiradas) pelo S.O. para recepção posterior
- Há a opção de não armazenarmos mensagens
  - Send com bloqueio: rendez-vous

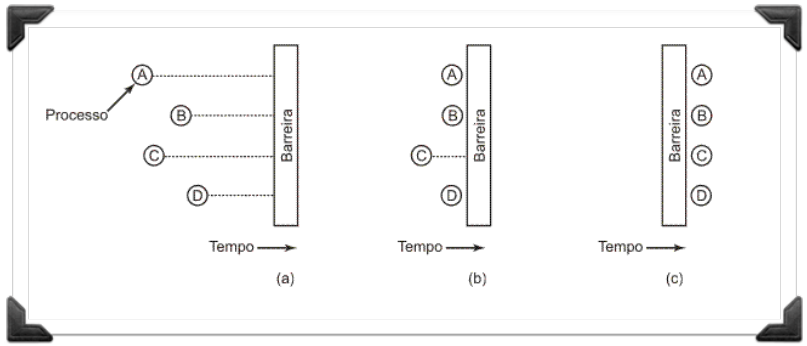
Troca de Mensagens

```
#define N 100                                   /* número de lugares no buffer */

void producer(void)
{
     int item;
     message m;                                 /* buffer de mensagens */

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* envia item para consumidor */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);                /* pega mensagem contendo item */
          item = extract_item(&m);              /* extrai o item da mensagem */
          send(producer, &m);                   /* envia a mensagem vazia como resposta */
          consume_item(item);                   /* faz alguma coisa com o item */
     }
}
```


Barreiras

- Manipulação de grupos de processos
  - Algumas aplicações são divididas em fases
  - Processos somente avançam depois que todos completaram a fase
- Barreira no final de cada fase
  - Primitiva barrier()

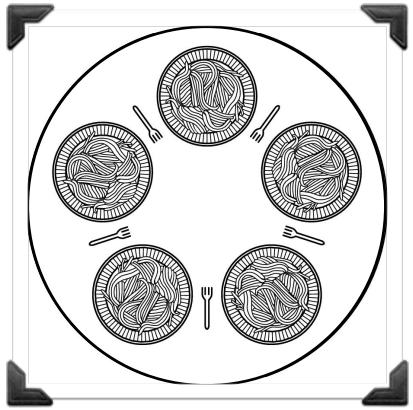
Barreiras



Problemas Clássicos de IPC

- **Problemas que ilustram situações que podem ocorrer no compartilhamento de recursos, que acabam sendo recorrentes em sistemas operacionais**
  - **Jantar dos Filósofos**
  - **Leitores e Escritores**
  - **Barbeiro Sonolento**


Jantar dos Filósofos



- **5 filósofos**
- **Filósofos passam o dia pensando ou comendo**
- **5 garfos, compartilhados**
- **São necessários dois garfos para comer**



Jantar dos Filósofos

```
#define N 5                         /* número de filósofos */

void philosopher(int i)             /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                   /* o filósofo está pensando */
        take_fork(i);               /* pega o garfo esquerdo */
        take_fork((i+1) % N);       /* pega o garfo direito; % é o operador modulo */
        eat( );                     /* hummm! Espaguete! */
        put_fork(i);                /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);        /* devolve o garfo direito à mesa */
    }
}
```

Não funciona! Deadlock!

Nem liberando garfo! Starvation! Livelock!


Jantar dos Filósofos

```
#define N           5           /* número de filósofos */
#define LEFT        (i+N-1)%N   /* número do vizinho à esquerda de i */
#define RIGHT       (i+1)%N     /* número do vizinho à direita de i */
#define THINKING    0           /* o filósofo está pensando */
#define HUNGRY      1           /* o filósofo está tentando pegar garfos */
#define EATING      2           /* o filósofo está comendo */
typedef int semaphore;          /* semáforos são um tipo especial de int */
int state[N];                   /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;            /* exclusão mútua para as regiões críticas */
semaphore s[N];                 /* um semáforo por filósofo */

void philosopher(int i)         /* i: o número do filósofo, de 0 a N-1 */
{
    while (TRUE) {              /* repete para sempre */
        think( );               /* o filósofo está pensando */
        take_forks(i);          /* pega dois garfos ou bloqueia */
        eat( );                 /* hummm! Espaguete! */
        put_forks(i);           /* devolve os dois garfos à mesa */
    }
}
```



Jantar dos Filósofos

```
void take_forks(int i)                  /* i: o número do filósofo, de 0 a N–1 */
{
     down(&mutex);                      /* entra na região crítica */
     state[i] = HUNGRY;                 /* registra que o filósofo está faminto */
     test(i);                           /* tenta pegar dois garfos */
     up(&mutex);                        /* sai da região crítica */
     down(&s[i]);                       /* bloqueia se os garfos não foram pegos */
}

void put_forks(i)                       /* i: o número do filósofo, de 0 a N–1 */
{
     down(&mutex);                      /* entra na região crítica */
     state[i] = THINKING;               /* o filósofo acabou de comer */
     test(LEFT);                        /* vê se o vizinho da esquerda pode comer agora */
     test(RIGHT);                       /* vê se o vizinho da direita pode comer agora */
     up(&mutex);                        /* sai da região crítica */
}

void test(i)                            /* i: o número do filósofo, de 0 a N–1 */
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```



Leitores e Escritores

- **Modela acessos a base de dados**
  - **Leitores podem acessar concorrentemente entre si**
  - **Escritores devem acessar de forma exclusiva**



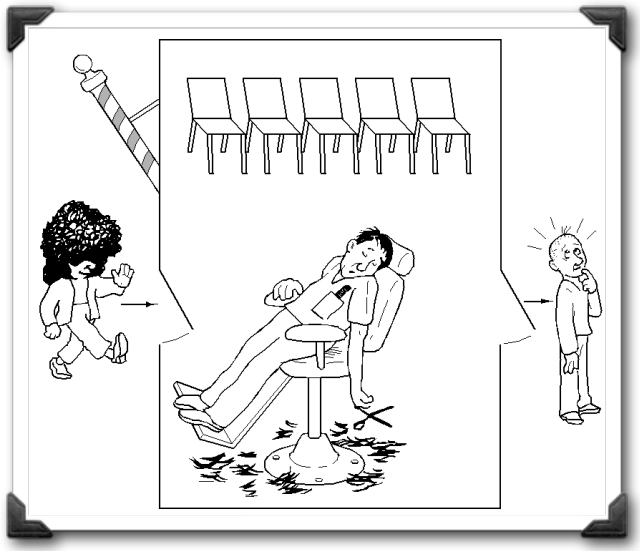
Leitores e Escritores

```
typedef int semaphore;                  /* use sua imaginação */
semaphore mutex = 1;                    /* controla o acesso a 'rc' */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* número de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                     /* repete para sempre */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc + 1;                  /* um leitor a mais agora */
          if (rc == 1) down(&db);       /* se este for o primeiro leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          read_data_base();             /* acesso aos dados */
          down(&mutex);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc – 1;                  /* um leitor a menos agora */
          if (rc == 0) up(&db);         /* se este for o último leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          use_data_read();              /* região não crítica */
     }
}

void writer(void)
{
     while (TRUE) {                     /* repete para sempre */
          think_up_data();              /* região não crítica */
          down(&db);                    /* obtém acesso exclusivo */
          write_data_base();            /* atualiza os dados */
          up(&db);                      /* libera o acesso exclusivo */
     }
}
```

Funciona, mas escritor pode ter que aguardar muito...



Barbeiro Sonolento





Barbeiro Sonolento

```
#define CHAIRS 5                    /* número de cadeiras para os clientes à espera */
typedef int semaphore;              /* use sua imaginação */
semaphore customers = 0;            /* número de clientes à espera de atendimento*/
semaphore barbers = 0;              /* número de barbeiros à espera de clientes */
semaphore mutex = 1;                /* para exclusão mútua */
int waiting = 0;                    /* clientes estão esperando (não estão cortando) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* vai dormir se o número de clientes for 0 */
        down(&mutex);               /* obtém acesso a 'waiting' */
        waiting = waiting – 1;      /* decresce de um o contador de clientes à espera */
        up(&barbers);               /* um barbeiro está agora pronto para cortar cabelo */
        up(&mutex);                 /* libera 'waiting' */
        cut_hair( );                /* corta o cabelo (fora da região crítica) */
    }
}

void customer(void)
{
    down(&mutex);                   /* entra na região crítica */
    if (waiting < CHAIRS) {         /* se não houver cadeiras livres, saia */
        waiting = waiting + 1;      /* incrementa o contador de clientes à espera */
        up(&customers);             /* acorda o barbeiro se necessário */
        up(&mutex);                 /* libera o acesso a 'waiting' */
        down(&barbers);             /* vai dormir se o número de barbeiros livres for 0 */
        get_haircut( );             /* sentado e sendo servido */
    } else {
        up(&mutex);                 /* a barbearia está cheia; não espere */
    }
}
```