

Sincronização

Parte I – Primitivas de Sincronização

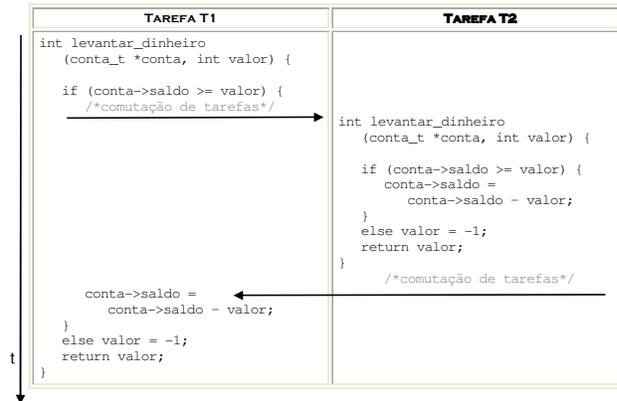
Problema da Exclusão Mútua

```
struct {
    int saldo;
    /* outras variáveis, ex. nome do titular, etc. */
} conta_t;

int levantar_dinheiro (conta_t* conta, int valor) {
    if (conta->saldo >= valor)
        conta->saldo = conta->saldo - valor;
    else
        valor = -1; /* -1 indica erro ocorrido */
    return valor;
}
```

- Problema se for multi-tarefa?

Problema da execução concorrente



8/9/2006

E se simplificarmos?...

```
struct {
    int saldo;
    /* outras variáveis, ex. nome do titular, etc. */
} conta_t;

int levantar_dinheiro (conta_t* conta, int valor) {
    conta->saldo = conta->saldo - valor;
    return valor;
}
```

Os programas em linguagem máquina têm acções mais elementares

;assumindo que a variável conta->saldo está na posição SALDO da memória

;assumindo que variável valor está na posição VALOR da memória

```
mov AX, SALDO ;carrega conteúdo da posição de memória  
                ;SALDO para registo geral AX  
mov BX, VALOR ;carrega conteúdo da posição de memória  
                ;VALOR para registo geral BX  
sub AX, BX    ;efectua subtracção AX = AX - BX  
mov SALDO, AX ;escreve resultado da subtracção na  
                ;posição de memória SALDO
```

8/9/2006

Secção crítica

Em programação concorrente sempre que se testam ou se modificam estruturas de dados partilhadas é necessário efectuar esses acessos dentro de uma secção crítica.

8/9/2006

Secção Crítica

```
int levantar_dinheiro (ref *conta, int valor)
{
    fechar(); /* lock() */
    if (conta->saldo >= valor) {
        conta->saldo = conta->saldo - valor;
    } else valor = -1
    abrir(); /* unlock */
    return valor;
}
```

} Secção crítica
(executada em
exclusão mútua)

Secção Crítica: Propriedades

- Exclusão mútua
- Progresso (liveness)
 - Ausência de interblocagem (deadlock)
 - Ausência de minguia (starvation)

Alocador de memória – exemplo de programação concorrente

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;

char* PedeMem() {
    ptr = pilha[topo];
    topo--;
    return ptr;
}

void DevolveMem(char* ptr) {
    topo++;
    pilha[topo]= ptr;
}
```

O programa está errado porque as estruturas de dados não são atualizadas de forma atômica

Alocador de Memória com secção crítica

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
trinco_t t = ABERTO;

char* PedeMem() {
    Fechar_mutex(mutex);
    ptr = pilha[topo];
    topo--;
    Abrir_mutex(mutex);
    return ptr;
}

void DevolveMem(char* ptr) {
    Fechar_mutex(mutex);
    topo++;
    pilha[topo]= ptr;
    Abrir_mutex(mutex);
}
```

Um trinco é criado sempre no estado ABERTO

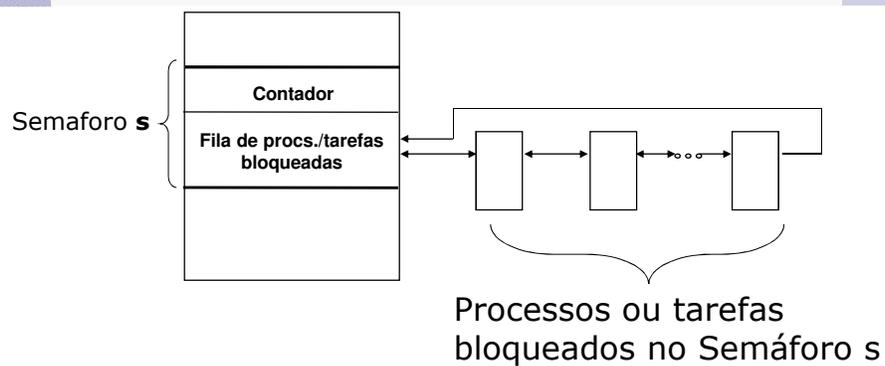
No início da secção crítica, os processos têm que chamar **Fechar_mutex**. Se o trinco estiver FECHADO, o processo espera que o processo abandone a secção crítica. Se estiver ABERTO, passa-o ao estado FECHADO. Estas operações executam-se **atOMICAMENTE**.

No fim da secção crítica, os processos têm que chamar **abrir_mutex**. Passa o trinco para o estado ABERTO ou desbloqueia um processo que esteja à sua espera de entrar na secção crítica

Trincos – Limitações

- Trincos não são suficientemente expressivos para resolver alguns problemas de sincronização
 - Ex: Bloquear tarefas se a pilha estiver cheia
 - Necessário um “contador de recursos” → só bloqueia se o número de pedidos exceder um limite

Semáforos



- Nunca fazer paralelo com semáforo de trânsito

Semáforos: Primitivas

- `s = criar_semaforo(num_unidades)`
 - cria um semáforo e inicializa o contador
- `esperar(s)`
 - bloqueia o processo se o contador for menor ou igual a zero; senão decrementa o contador
- `assinalar(s)`
 - se houver processos bloqueados, liberta um; senão, incrementa o contador
- Todas as primitivas se executam atómicamente
- `esperar()` e `assinalar()` podem ser chamadas por processos diferentes.

Como usar semáforos para gerir conjunto limitado de recursos partilhados

- Um processo requisita um recurso
 - Executa `Esperar(SemRecursos)`
- Um processo liberta um recurso
 - Executa `Assinalar(SemRecurso)`
- O semáforo que controla o algoritmo é inicializado com o número de recursos disponíveis
 - `SemRecurso = CriarSemaforo(NUM_RECURSOS)`

Semáforos (exemplo)

Alocador de memória com O semáforo SemMem a controlar a existência de memória livre

```
#define MAX_PILHA 100
char* pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
semáforo_t SemMem;
semáforo_t mutex;

char* PedeMem() {
    Esperar(SemMem);
    Fechar(mutex);
    ptr = pilha[topo];
    topo--;
    Sair(mutex);
    return ptr;
}

void DevolveMem(char ptr) {
    Fechar(mutex);
    topo++;
    pilha[topo]= ptr;
    Sair(mutex);
    Assinalar(SemMem);
}

main() {
    /*...*/
    mutex = CriarMutex();
    semMem = CriarSemaforo(MAX_PILHA);
}
```

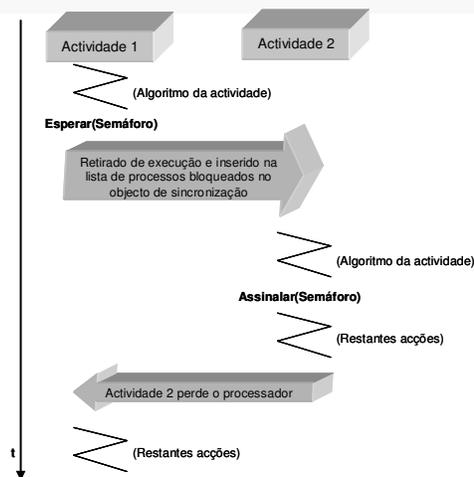
>?

Interblocagem
A troca das operações sobre os semáforos criaria uma situação de erro

O semáforo é inicializado com o valor dos recursos disponíveis

8/9/2006

Situação mais simples de cooperação



8/9/2006

Cooperação entre dois processos com semáforos

```
Proci
semEvent = CriarSemaforo(0);
void EsperarAcontecimento() {
    Esperar (semEvent);
}

Procj
void AssinalarAcontecimento() {
    Assinalar (semEvent);
}
```

**O semáforo é
inicializado a
zero**

8/9/2006

Semáforos: Variantes

- Genérico: `assinalar()` liberta um processo qualquer da fila
- FIFO: `assinalar()` liberta o processo que se bloqueou há mais tempo
- Semáforo com prioridades: o processo especifica em `esperar()` a prioridade, `assinalar()` liberta os processos por ordem de prioridades
- Semáforo com unidades: as primitivas `esperar()` e `assinalar()` permitem especificar o número de unidades a esperar ou assinalar

Interface POSIX: Mutexes

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                      pthread_mutexattr_t *attr);
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_timedlock(pthread_mutex_t *mutex,
                           const struct timespec *timeout);
```

Exemplo:

```
pthread_mutex_t count_lock;

pthread_mutex_init(&count_lock, NULL);
pthread_mutex_lock(&count_lock);
count++;
pthread_mutex_unlock(&count_lock);
```

```
#include <stdio.h>
#include <pthread.h>
#define MAX_PILHA 100
char * pilha[MAX_PILHA];
int topo = MAX_PILHA-1;
pthread_mutex_t semExtMut; /* semáforo exclusão mútua */

char* PedeMem()
{
    char* ptr;

    pthread_mutex_lock(&semExtMut);
    if (topo >= 0) { /* verificar se há memória livre */
        ptr = pilha[topo]; /* devolve bloco livre */
        topo--;
    }
    else {ptr = NULL /* indica erro */}
    pthread_mutex_unlock(&semExtMut);
    return ptr;
}

void DevolveMem(char * ptr)
{
    pthread_mutex_lock(&semExtMut);
    topo++;
    pilha[topo] = ptr;
    pthread_mutex_unlock(&semExtMut);
}

int main(int argc, char *argv[]) {
    pthread_mutex_init(&semExtMut, NULL); /* attr default */
    /* programa */
    pthread_mutex_destroy(&semExtMut);
    return 0;
}

8/9/2006
```

**Alocador com
secção crítica
Programado
com mutex
da biblioteca
Posix**

Interface POSIX: Semáforos

```
int sem_init(sem_t *sem, int pshared, unsigned value);  
int sem_post(sem_t *sem);  
int sem_wait(sem_t *sem);
```

Exemplo:

```
sem_t sharedsem;  
sem_init(&sharedsem, 0, 1);  
sem_wait(&sharedsem);  
count++;  
sem_post(&sharedsem);
```

Secção Crítica: Implementações

- Algorítmicas
- Hardware
- Sistema Operativo

Secção Crítica: Propriedades

- Exclusão mútua
- Progresso (liveness)
 - Ausência de interbloqueamento (deadlock)
 - Ausência de minguagem (starvation)

Proposta #1

```
int trinco = ABERTO;

fechar() {
    while (trinco == FECHADO) ;
    trinco = FECHADO;
}

abrir() {
    trinco = ABERTO;
}
```

Qual a propriedade
que não é garantida?

Proposta #2

```
int vez = 1;

t1_fechar() {
    while (vez == 2) ;
}

t1_abrir() {
    vez = 2;
}

t2_fechar() {
    while (vez == 1) ;
}

t2_abrir() {
    vez = 1;
}
```

Qual a propriedade
que não é garantida?

Proposta #3

```
int t1_quer_entrar = FALSE, t2_quer_entrar = FALSE;

t1_fechar() {
    while (t2_quer_entrar == TRUE) ;
    t1_quer_entrar = TRUE;
}

t1_abrir() {
    t1_quer_entrar = FALSE;
}

/* t2 -> simetrico */
```

Qual a propriedade
que não é garantida?

Proposta #4

```
int t1_quer_entrar = FALSE, t2_quer_entrar = FALSE;

t1_fechar() {
    t1_quer_entrar = TRUE;
    while (t2_quer_entrar == TRUE) ;
}

t1_abrir() {
    t1_quer_entrar = FALSE;
}

/* t2 -> simetrico */
```

Porque motivo
é garantida a
exclusão mútua?

Qual a propriedade
que não é garantida?

Algoritmo Peterson

```
int t1_quer_entrar = FALSE;
int t2_quer_entrar = FALSE;
int tar_prio = 1;

t1_fechar()
{
    t1_quer_entrar = TRUE;
    tar_prio = 2;
    while (t2_quer_entrar &&
           (tar_prio == 2))
        ;
}

t1_abrir()
{
    t1_quer_entrar = FALSE;
}

t2_fechar()
{
    t2_quer_entrar = TRUE;
    tar_prio = 1;
    while (t1_quer_entrar &&
           (tar_prio == 1))
        ;
}

t2_abrir()
{
    t2_quer_entrar = FALSE;
}
```

Porque motivo
é garantida a
ausência de
míngua?

Algoritmo de Bakery

```

int choosing[N]; // Inicializado a FALSE
int ticket[N]; // Inicializado a 0
fechar(int i)
{
    int j;
    choosing[i] = TRUE;
    ticket[i] = 1 + maxn(ticket);
    choosing[i] = FALSE;

    for (j=0; j<N; j++) {
        if (j==i) continue;
        while (choosing[j]) ;

        while (ticket[j]!=0 &&
              ((ticket[i] > ticket[j]) ||
               (ticket[i] == ticket[j] && i > j)))
            ;
    }
}
abrir(int i)
{
    ticket[i] = 0;
}

```

•Pi indica que está a escolher a senha
•Escolhe uma senha maior que todas as outras
•Anuncia que escolheu já a senha

•Pi verifica se tem a menor senha de todos os Pjb

•Se Pj estiver a escolher uma senha, espera que termine

•Neste ponto, Pj ou já escolheu uma senha, ou ainda não escolheu
•Se escolheu, Pi vai ver se é menor que a sua
•Se não escolheu, vai ver a de Pi e escolher uma senha maior

•Se o ticket de Pi for menor, Pi entra
•Se os tickets forem iguais, entra o que tiver o menor identificador

Conclusões sobre as Soluções Algorítmicas

- Complexas → Latência
- Atenção a arquiteturas em que cada tarefa acede a uma cache diferente!
 - Escritas e leituras feitas por diferentes tarefas podem chegar em ordens diferentes às outras tarefas e tornar algoritmos incorrectos
 - Normalmente resolve-se obrigando as caches a serem descarregadas antes/depois de acessos partilhados
- Só contemplam **espera activa**
- Solução: Introduzir instruções hardware para facilitar a solução

Soluções com suporte do hardware

- Abrir() e Fechar() usam instruções especiais oferecidas pelos processadores
 - Inibição de interrupções
 - Exchange (xchg no Intel IA)
 - Test-and-set (cmpxchg no Intel IA)

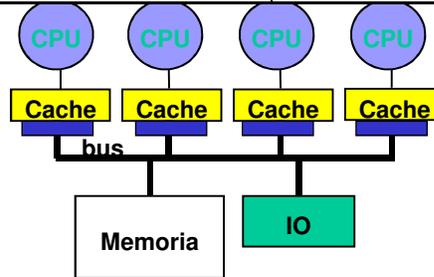
Exclusão mútua com inibição de interrupções

```
int mascara;  
  
Fechar_hard () {  
  mascara = mascarar_int ();  
}  
  
Abrir_hard () {  
  repoe_int (mascara);  
}
```

- Este mecanismo só deve ser utilizado dentro do sistema operativo em secções críticas de muito curta duração
 - Inibição das interrupções impede que se executem serviços de sistema (I/O, etc)
 - Se o programa se esquecer de chamar abrir(), as interrupções ficam inibidas e o sistema fica parado
- Não funciona em multiprocessadores

O problema da atomicidade com multiprocessadores

	P1	P2
Instante 1	P1 inibe as suas interrupções e entra na secção crítica	
Instante 2		P2 inibe as suas interrupções e entra na secção crítica
Instante 3	P1 na secção crítica ERRO!!	P2 na secção crítica ERRO!!



Exclusão mútua com exchange

```

int trinco = #0;

fechar()
{
    li R1, trinco ;      guarda em R1 o endereço do trinco
    li R2, #1
l1:  exch R2, [trinco];  faz o exchange
    bnez R2, l1 ;       se estava a 0 entra, senão repete
}

abrir()
{
    trinco = FALSE;
}

```

Exclusão mútua com exchange

```

int trinco = FALSE;

fechar()
{
    li R1, trinco ;
    li R2, #1
l1:  exch R2, 0(R1);
    bnez R2, l1 ;
}

abrir()
{
    trinco = FALSE;
}

```

•O exchange corresponde às seguintes operações executadas de forma atômica

- lw Rt, 0(R1)
- sw 0(R1), R2
- mov R2, Rt

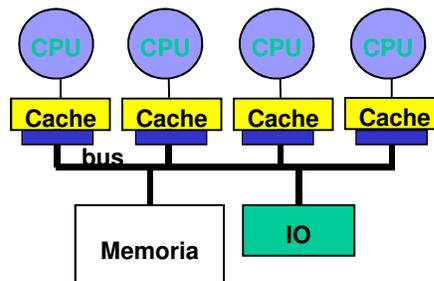
•A atomicidade é conseguida mantendo o bus trancado entre o Load e o Store

•R2 contém o valor que estava no trinco

- Se era 0, o trinco estava livre
 - O processo entra
 - O exchange deixou o trinco trancado
- Se não era 0, significa que o trinco estava trancado
 - O processo fica em espera activa até que encontre o trinco aberto

Exchange em multiprocessadores

	P1	P2
Instante 1	P1 inicia exchange e tranca o bus	
Instante 2	P1 completa exchange e tranca a secção crítica	P2 tenta fazer exchange mas bloqueia-se a tentar obter o bus
Instante 3	P1 entra secção crítica	P2 verifica que o trinco está trancado e fica em espera activa



Exclusão mútua com test-and-set

```

int trinco = FALSE;

fechar()
{
    li R1, trinco;           guarda em R1 o endereço do trinco
    ll: tst R2, 0(R1);       faz o test-and-set
    bnez R2, ll ;           se não estava set entra, senão repete
}

abrir()
{
    trinco = FALSE;
}

```

Exclusão mútua com test-and-set

```

int trinco = FALSE;

fechar()
{
    li R1, trinco;           guarda em R1 o endereço do trinco
    ll: tst R2, 0(R1);       faz o test-and-set
    bnez R2, ll ;           se não estava set entra, senão repete
}

abrir()
{
    trinco = FALSE;
}

```

- O test-and-set corresponde às seguintes operações executadas de forma atômica
 - lw R2, 0(R1)
 - sw 0(R1), #1
- A atomicidade é conseguida mantendo o bus trancado entre o Load e o Store

- R2 contém o valor que estava no trinco
 - Se era 0, o trinco estava livre
 - O processo entra
 - O test-and-set deixou o trinco trancado
 - Se não era 0, significa que o trinco estava trancado
 - O processo fica em espera activa até que encontre o trinco aberto

Conclusões sobre as Soluções com Suporte do Hardware

- Oferecem os mecanismos básicos para a implementação da exclusão mútua, mas...
- Algumas não podem ser usadas directamente por programas em modo utilizador
 - E.g., inibição de interrupções
- Outras só contemplam espera activa
 - E.g., exchange, test-and-set

Soluções com Suporte do SO

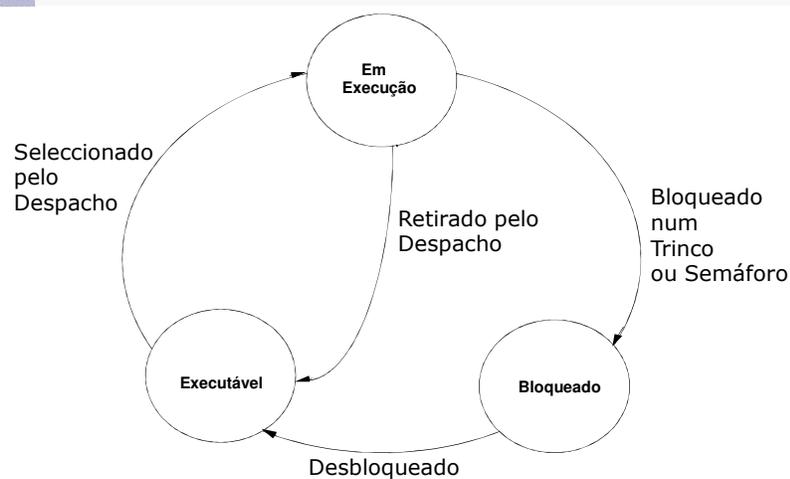
- Primitivas de sincronização são chamadas ao SO
 - Software trap (interrupção SW)
 - Comutação para modo núcleo
 - Estruturas de dados e código de sincronização presente ao núcleo
 - Usa o suporte de hardware (exch./test-and-set/inibição de interrupções)
- Veremos duas primitivas de sincronização
 - Trincos
 - Semáforos

Soluções com Suporte do SO: Vantagens

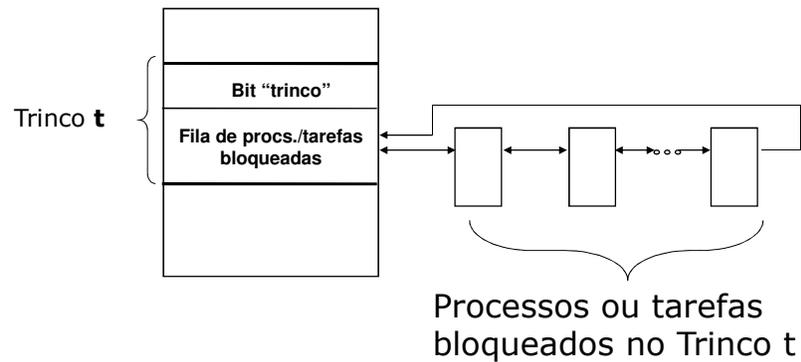
- Se SO souber quais processos estão à espera de secção crítica, nem sequer lhes dá tempo de processador
 - Minimiza-se a espera activa
- Soluções hardware podem ser usadas, mas pelo código do SO
 - Dentro das chamadas sistema que suportam o abrir(), fechar(), etc.

8/9/2006

Diagrama de Estado dos Processos / Tarefas



Estruturas de dados associados aos Trincos



Funções do trinco (*mutex*)

```
t.var=ABERTO; // trinco inicialmente ABERTO
t.numTarefasBloqueadas = 0;
```

```
Fechar_mutex (trinco_t t) {
    if (t.var == FECHADO) {
        numTarefasBloqueadas++;
        bloqueia_tarefa();
    }
    else {
        t.var = FECHADO;
    }
}
```

```
Abrir_mutex (trinco_t t) {
    if (numTarefasBloqueadas > 0) {
        desbloqueia_tarefa();
        numTarefasBloqueadas--;
    }
    else t.var = ABERTO;
}
```

- Retirar tarefa de execução
- Salvar o seu contexto
- Marcar o seu estado como bloqueada
- Colocar a estrutura de dados que descreve a tarefa na fila de espera associada ao trinco
- Invocar o algoritmo de escalonamento

- Existem tarefas bloqueadas
- Marcar o estado de uma delas como "executável"
- Retirar a estrutura de dados que descreve a tarefa da fila de espera associada ao trinco

- Este programa concorrente está errado!
- É necessário assegurar que variáveis partilhadas são acedidas em exclusão mútua

Funções do trinco (*mutex*)

```
trinco_t = t;
t.var=ABERTO; // trinco inicialmente ABERTO
t.numTarefasBloqueadas = 0;
```

```
Fechar_mutex (trinco_t t) {
  Fechar_hard(t);
  if (t.var == FECHADO){
    numTarefasBloqueadas++;
    Abrir_hard(t);
    bloqueia_tarefa();
  }
  else {
    t.var = FECHADO;
    Abrir_hard(t);
  }
}
```

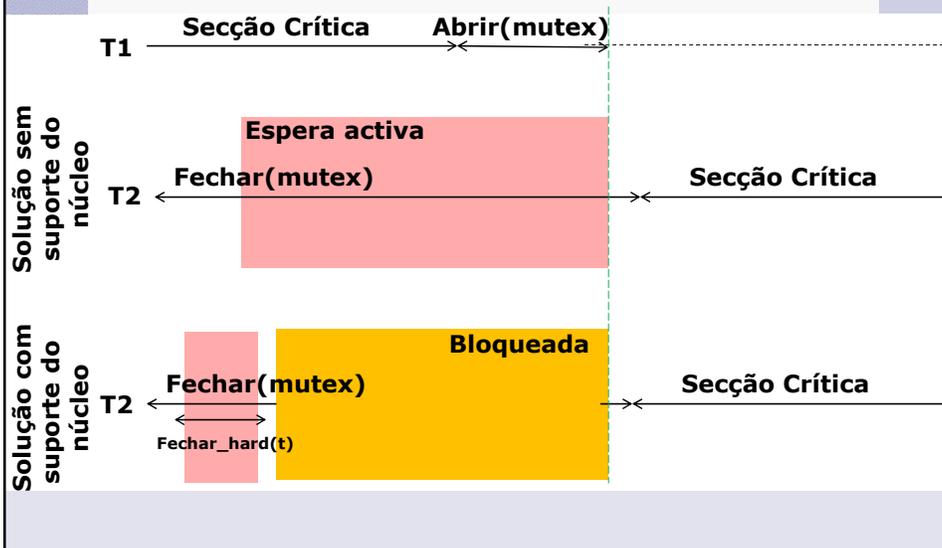
•É necessário assegurar exclusão mútua no acesso aos atributos do trinco

Qual a diferença entre a exclusão mútua no acesso aos atributos do trinco e a exclusão mútua que o trinco assegura?

```
Abrir_mutex (trinco_t t) {
  Fechar_hard(t);
  if (numTarefasBloqueadas > 0)
    {desbloqueia_tarefa();
     numTarefasBloqueadas--;}
  else t.var = ABERTO;
  Abrir_hard(t)
}
```

8/9/2006

Espera activa com e sem suporte do núcleo



Semáforos: Primitivas

```
typedef struct {  
    int contador;  
    queue_t  
    fila_procs;  
} semaforo_t;
```

