


Trinco do Sistema Operativo

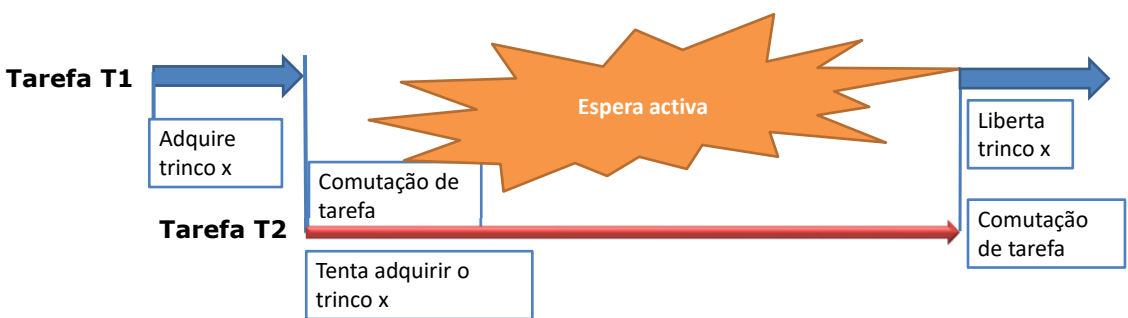
Sistemas Operativos – DEI - IST

1



Recapitulação das soluções anteriores

- Inibir interrupções → não é possível em modo utilizador
- Soluções algorítmicas → complexas e elevada latência
- Mecanismo de espera → espera activa



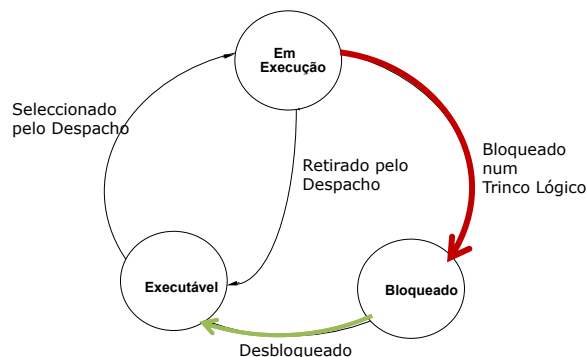
Sistemas Operativos – DEI - IST

2



Eliminar a Espera Activa com suporte do Sistema Operativo

- Retirar a tarefa da fila do Despacho sempre que não pode prosseguir a sua execução
- Corresponde a mudar o estado da tarefa para **Bloqueado**



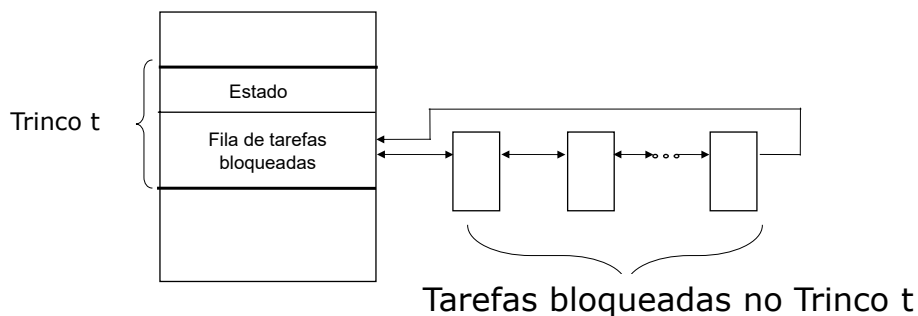
Sistemas Operativos – DEI - IST

3



Mutex com suporte do núcleo

- O *mutex* tem um estado associado ABERTO ou FECHADO
- Caso a tarefa tente adquirir um *mutex* Fechado, o núcleo retira-a de execução, bloqueando-a numa lista de tarefas bloqueadas associadas ao *mutex*



Sistemas Operativos – DEI - IST

4



Objecto Mutex

Propriedades

Identificador

Estado

Lista das tarefas
bloqueadas

Operações

CriarMutex

Fechar (lock)

Abrir (unlock)

EliminarMutex

Sistemas Operativos – DEI - IST

5



Funções *lock* e *unlock*

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

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

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

1ª versão

Sistemas Operativos – DEI - IST

6



Semântica do *unlock*

- Quando um trinco é libertado e existem tarefas bloqueadas, uma vai poder continuar
- A tarefa que se vai executar quando há várias bloqueadas é normalmente indeterminada, ou seja, na maioria das implementações não se assume uma politica na libertação da tarefa (por exemplo FIFO)

Sistemas Operativos – DEI - IST

7



Interface POSIX para mutexes

- Na definição das *threads* POSIX a interface dos objectos de sincronização foi também englobada
- As funções de *lock* e *unlock* têm uma interface evidente

```
int pthread_mutex_lock(pthread_mutex_t *mutex)

int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Sistemas Operativos – DEI - IST

8



Inicialização dos *mutexes* POSIX

- Existem duas formas de inicialização
- Declaração de um *mutex* como uma variável estática

```
pthread_mutex_t trincoA = PTHREAD_MUTEX_INITIALIZER
```

- Declaração dinâmica durante a execução

```
int pthread_mutex_init(pthread_mutex_t *trincoA, pthread_mutexattr_t *attr)
```

Identificador
do mutex

Atributos previamente
definidos pode ser NULL

Sistemas Operativos – DEI - IST

9



Incrementar uma variável global correctamente programada

```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

for (j = 0; j < loops; j++) {
    s = pthread_mutex_lock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_lock");

    loc = glob;
    loc++;
    glob = loc;

    s = pthread_mutex_unlock(&mtx);
    if (s != 0)
        errExitEN(s, "pthread_mutex_unlock");
}
```

Sistemas Operativos – DEI - IST

11

Programar com Objetos partilhados

- Até agora, aprendemos esta receita para programar concorrentemente com memória partilhada:
 - Identificar secções críticas
 - Sincronizar cada secção crítica com um *mutex*
- Um trinco global ou múltiplos trincos finos?
- Preciso mesmo usar trinco?

12

Quando preciso de usar *mutexes*?

```

struct {
    int saldo;
    int cliente;
    ...
} conta_t;
conta_t contas[N] = ...; //Mantém as contas todas do banco

int transferir_dinheiro(conta_t *a, conta_t *b, valor) {...}

int saldoTotalDeCliente(int cliente) {
    int total = 0;
    for (int i=0; i<N; i++) {
        if (contas[i].cliente == cliente) {
            total += contas[i].saldo;
        }
    }
    return total;
}

```

Se esta função só lê, é mesmo necessário usar um trinco?

Não usar trinco: arriscamos resultados inconsistentes!

Contudo, é um problema para reflectir que *mutexes* necessitam para garantir a consistência: conta a conta ou a totalidade das contas?

13

Como sincronizar esta função?

```

struct {
    int saldo;
    ...
} conta_t;

int levantar_dinheiro (conta_t* conta, int valor) {

    pthread_mutex_lock(&conta.t);

    if (conta->saldo >= valor)
        conta->saldo = conta->saldo - valor;
    else
        valor = -1; /* -1 indica erro ocorrido */

    pthread_mutex_unlock(&conta.t);
    return valor;
}

```

14



Função levantamento bancário

```

struct {
    int saldo;
    static pthread_mutex_t t = PTHREAD_MUTEX_INITIALIZER;
    /* outras variáveis, ex. nome do titular, etc. */
} conta_t;

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

```

Sistemas Operativos – DEI - IST

15

Trinco global

- Normalmente é a solução mais simples
- Mas limita o paralelismo
 - Quanto mais paralelo for o programa, maior é a limitação
- Exemplo: “big kernel lock” do Linux
 - Criado nas primeiras versões do Linux (versão 2.0)
 - Grande barreira de escalabilidade
 - Finalmente removido na versão 2.6

17

Trincos finos: programação com objetos partilhados

- Objeto cujos métodos podem ser chamados em concorrência por diferentes tarefas devem ter:
 - Interface dos métodos públicos
 - Código de cada método
 - Variáveis de estado
 - **Variáveis de sincronização**
 - **Um trinco para garantir que métodos críticos se executam em exclusão mútua**
 - Opcionalmente: semáforos, variáveis de condição

18

Programar com trincos finos

- Em geral, maior paralelismo
- **Mas pode trazer *bugs* difíceis de resolver...**

19

Exemplo com trincos finos

```
transferir(conta a, conta b, int montante) {  
    fechar(a.trinco);  
    debitar(a, montante);  
    fechar(b.trinco);  
    creditar(b, montante);  
    abrir(a.trinco);  
    abrir(b.trinco);  
}
```

O que pode correr mal?

21



Conclusão

- No ambiente multitarefa a utilização de *mutexes* permite programar corretamente as secções críticas necessárias.
- Como o núcleo evita a espera ativa otimizam o desempenho do sistema
- Como são funções que recorrem ao núcleo tem um impacto no desempenho, mas é um preço pequeno a pagar