Sistemas Operacionais

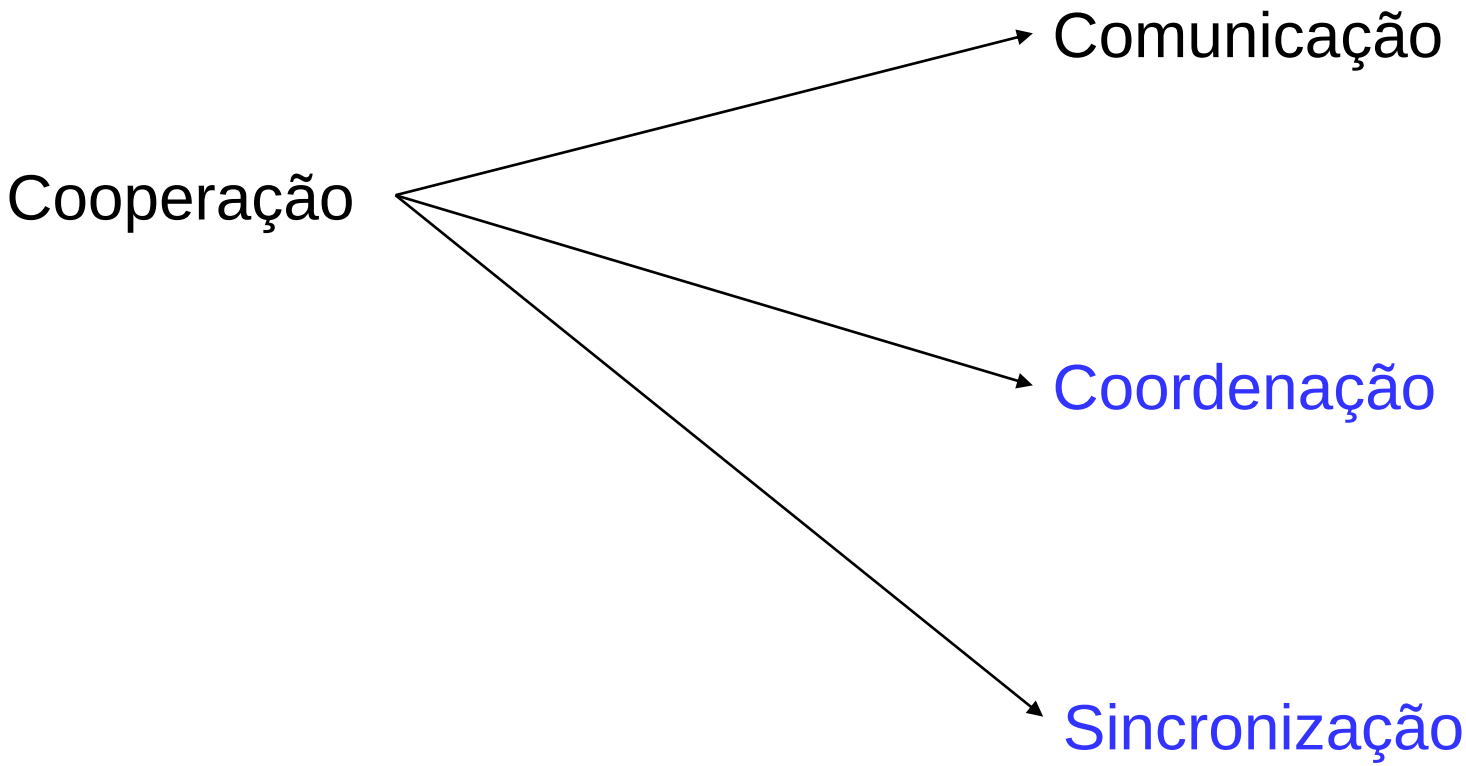
Coordenação de Tarefas

Coordenação de Tarefas

Índice

- Tarefas cooperativas
- Comunicação, coordenação e sincronização
- Condições de disputa
  - Exemplo: depósito bancário
- Seção crítica
- Soluções do tipo Espera Ocupada

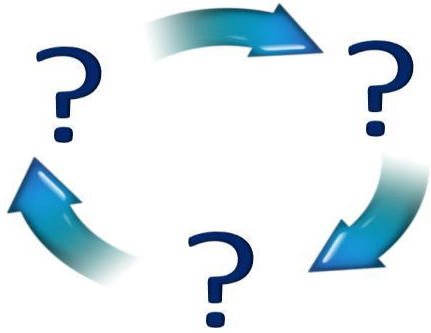
Coordenação de Tarefas



Coordenação de Tarefas

Cooperação

Comunicação



Coordenação de Tarefas

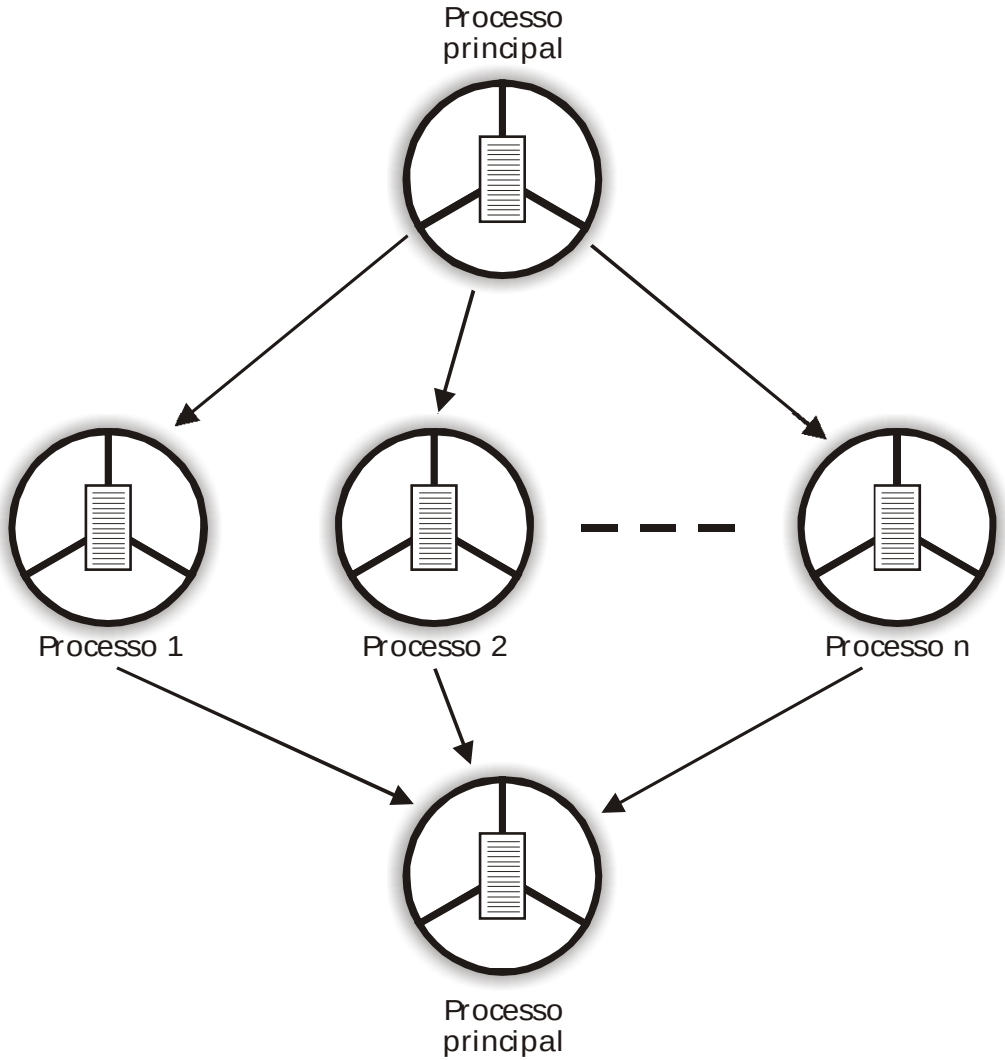
Comunicação: Compartilhamento de informações necessárias à execução de cada tarefa.

Coordenação: Das atividades para que os resultados sejam coerentes (sem erros).

Sincronização: O acesso ao recurso compartilhado ou execução da tarefa exige a sincronização (às vezes vinculada a uma condição).

Concorrência

```
PARBEGIN
    Comando_1;
    Comando_2;
        .
        .
    Comando_n;
PAREND
```



Condições de Disputa

Acesso concorrente a recursos compartilhados

- **Memória (dados)**
- **Arquivos**
- **Conexões de rede**
- **...**

⇒ Problemas de consistência dos *dados* ou do *estado* do recurso.

Condições de Disputa

I Am Devloper
@iamdevloper

Knock knock
Race condition
Who's there?

Exemplo: Transação Bancária

Transação bancária

Exemplo: Operação bancária

```
.
.
typedef struct conta_t
{
        int saldo ;        // saldo atual da conta
        ...                // outras informações da conta
} conta_t ;

void depositar (conta_t* conta, int valor)
{
        conta->saldo += valor ;
}
```

Transação bancária

Código montado para plataforma x86:

00000000 <depositar>:

```
push %ebp                 # guarda o valor do "stack frame“ (pilha)
mov %esp,%ebp             # ajusta o stack frame para executar a função
mov 0x8(%ebp),%ecx        # mem(saldo) → reg1
mov 0x8(%ebp),%edx        # mem(valor) → reg2
add (%edx),%eax           # reg1 + reg2 → reg1
mov %eax,(%ecx)           # reg1 → mem(saldo)
leave                     # restaura o stack frame anterior
ret                       # retorna à função anterior
```

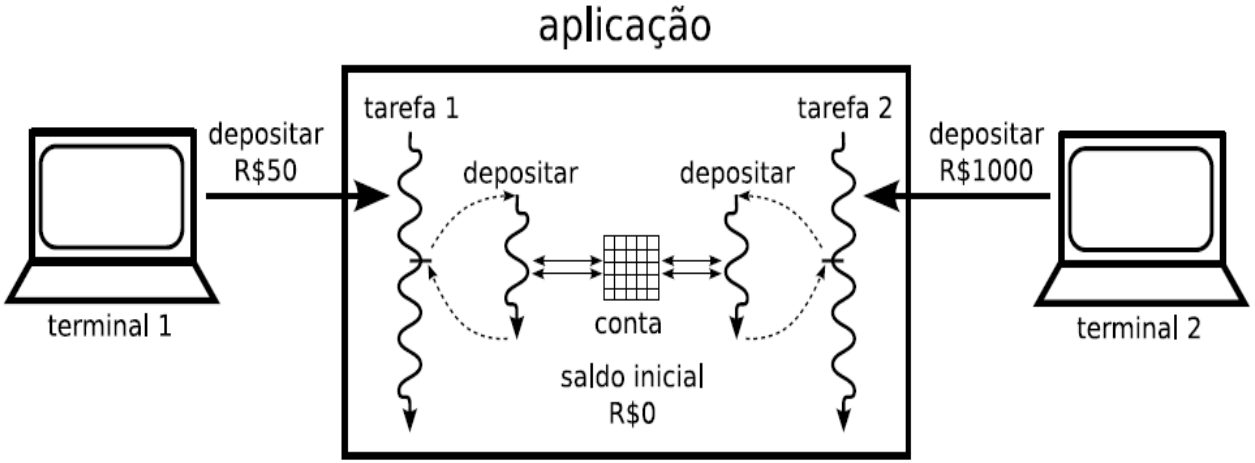
mem(x) ⇒ posição de memória

reg_i ⇒ registrador

Transação bancária

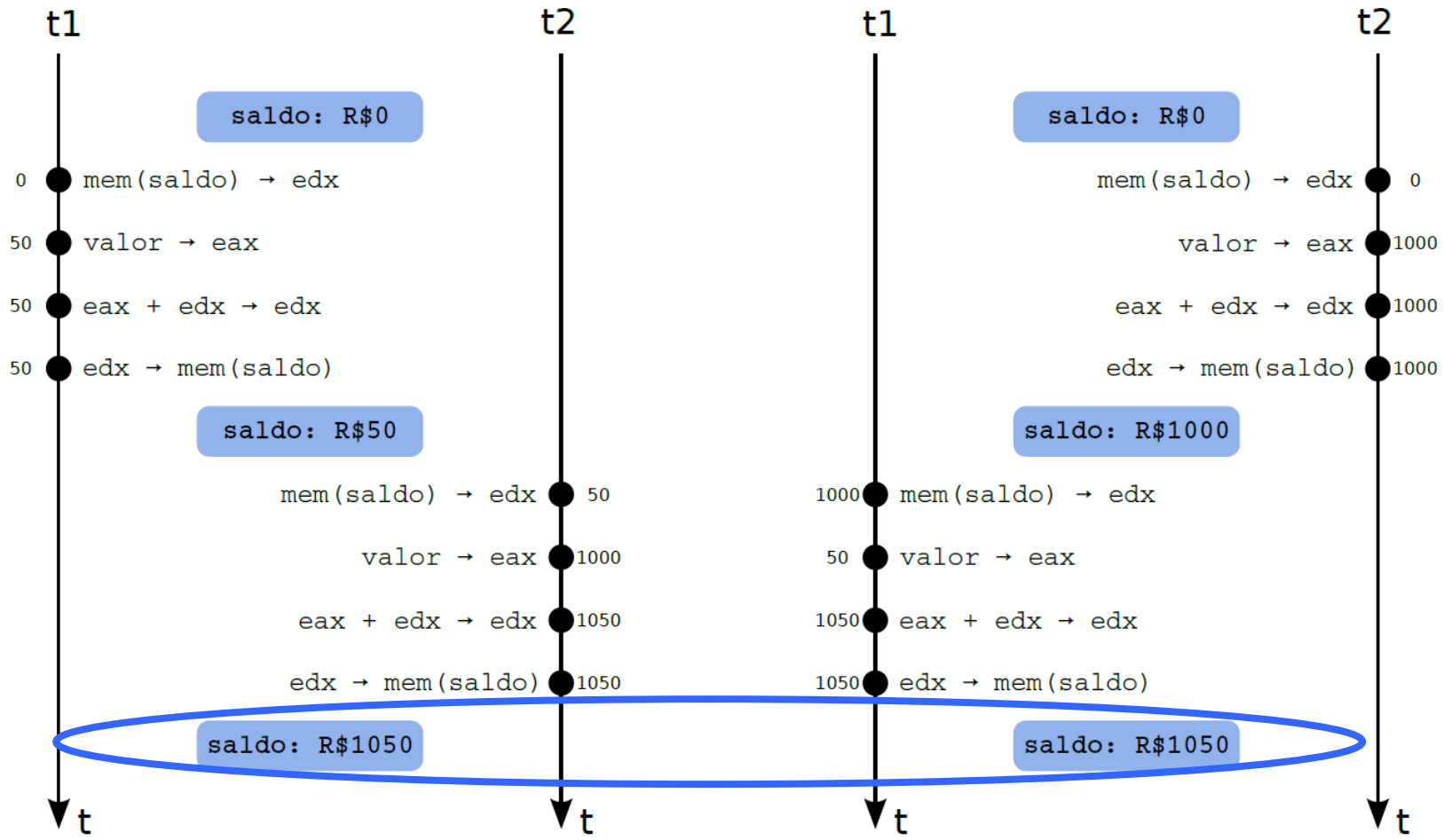
Função *depositar* faz parte de um sistema de controle de contas que pode ser acessado simultaneamente por *inúmeros* usuários.

Ex.: dois usuários tentam fazer depósito *na mesma conta ao mesmo tempo*:



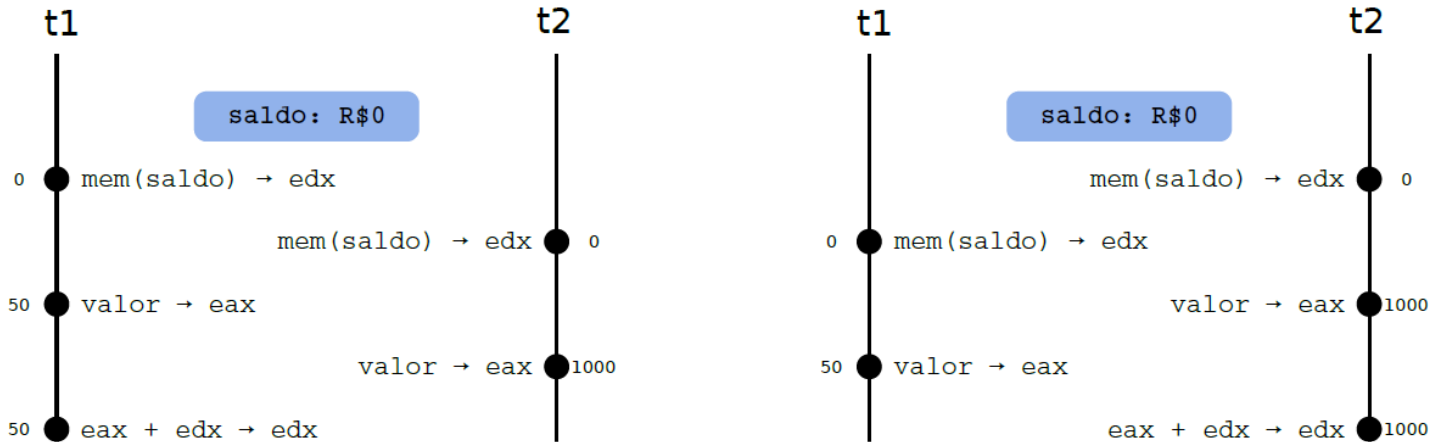
Comportamento dinâmico

1 – t_1 é executada integralmente antes ou depois de t_2



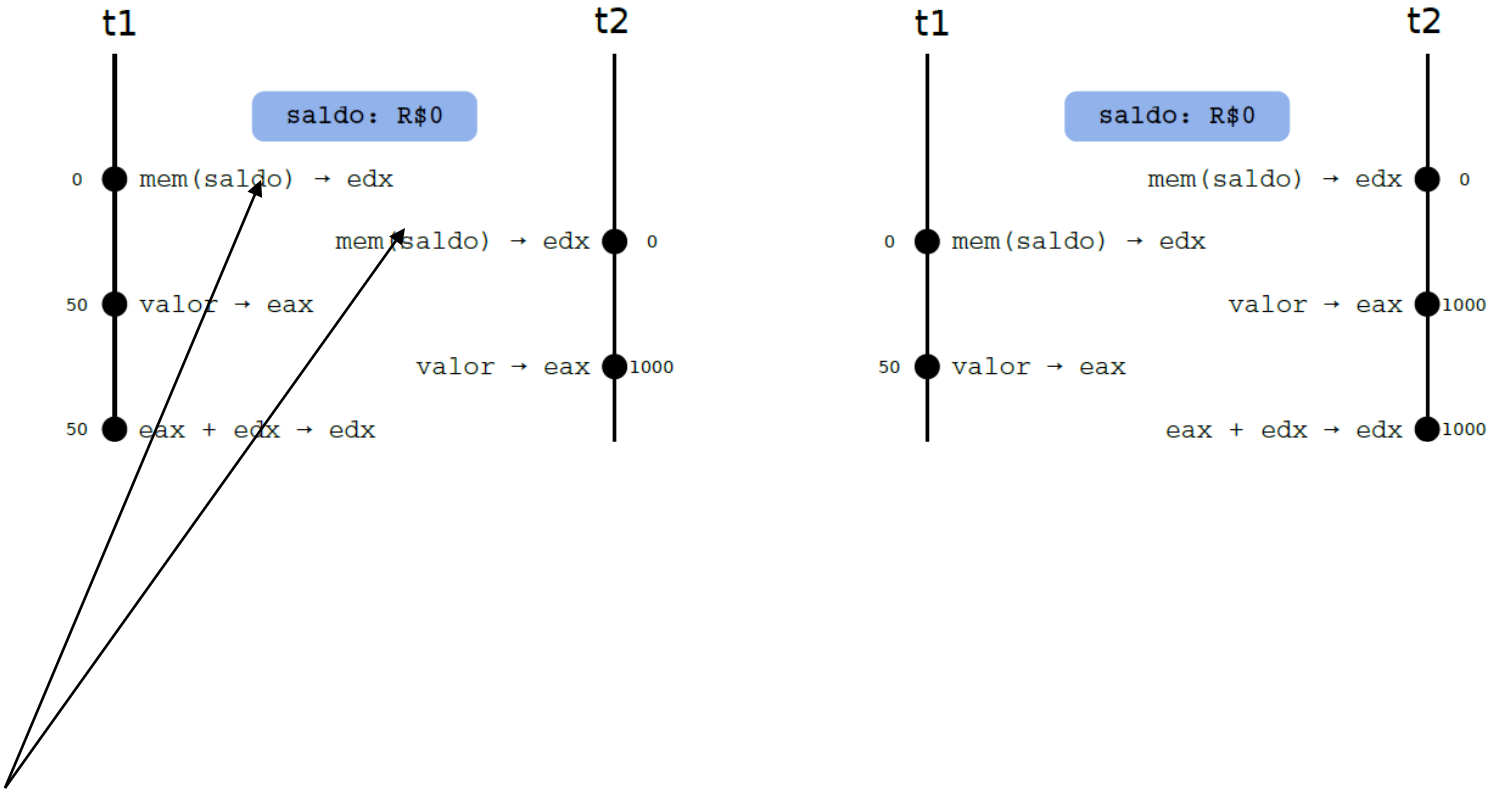
Comportamento dinâmico

2 – t_1 e t_2 se entrelaçam



Comportamento dinâmico

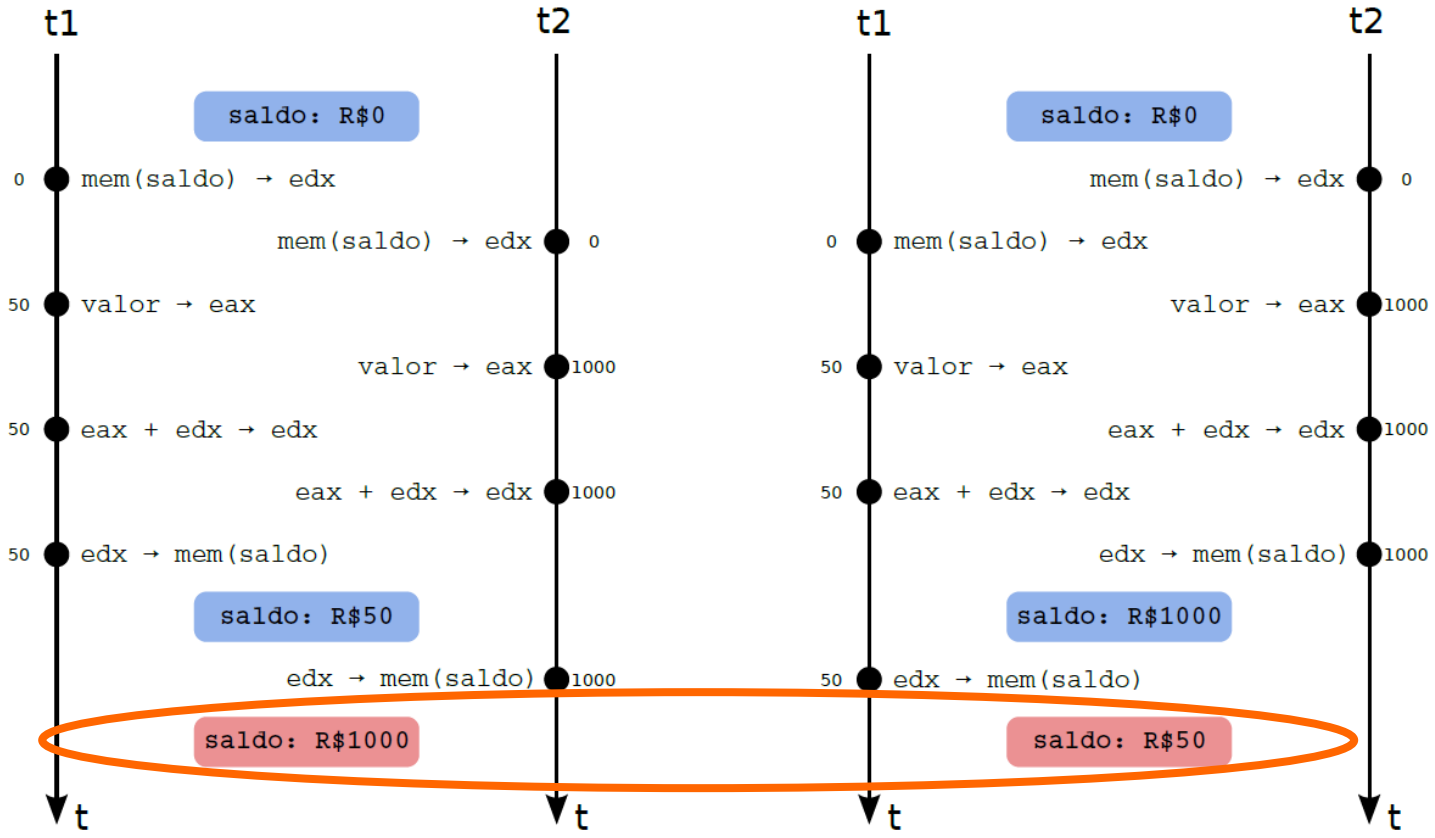
2 – t_1 e t_2 se entrelaçam



Obs.: Os registradores são salvos/restaurados durante a troca de contexto.

Comportamento dinâmico

2 – t_1 e t_2 se entrelaçam



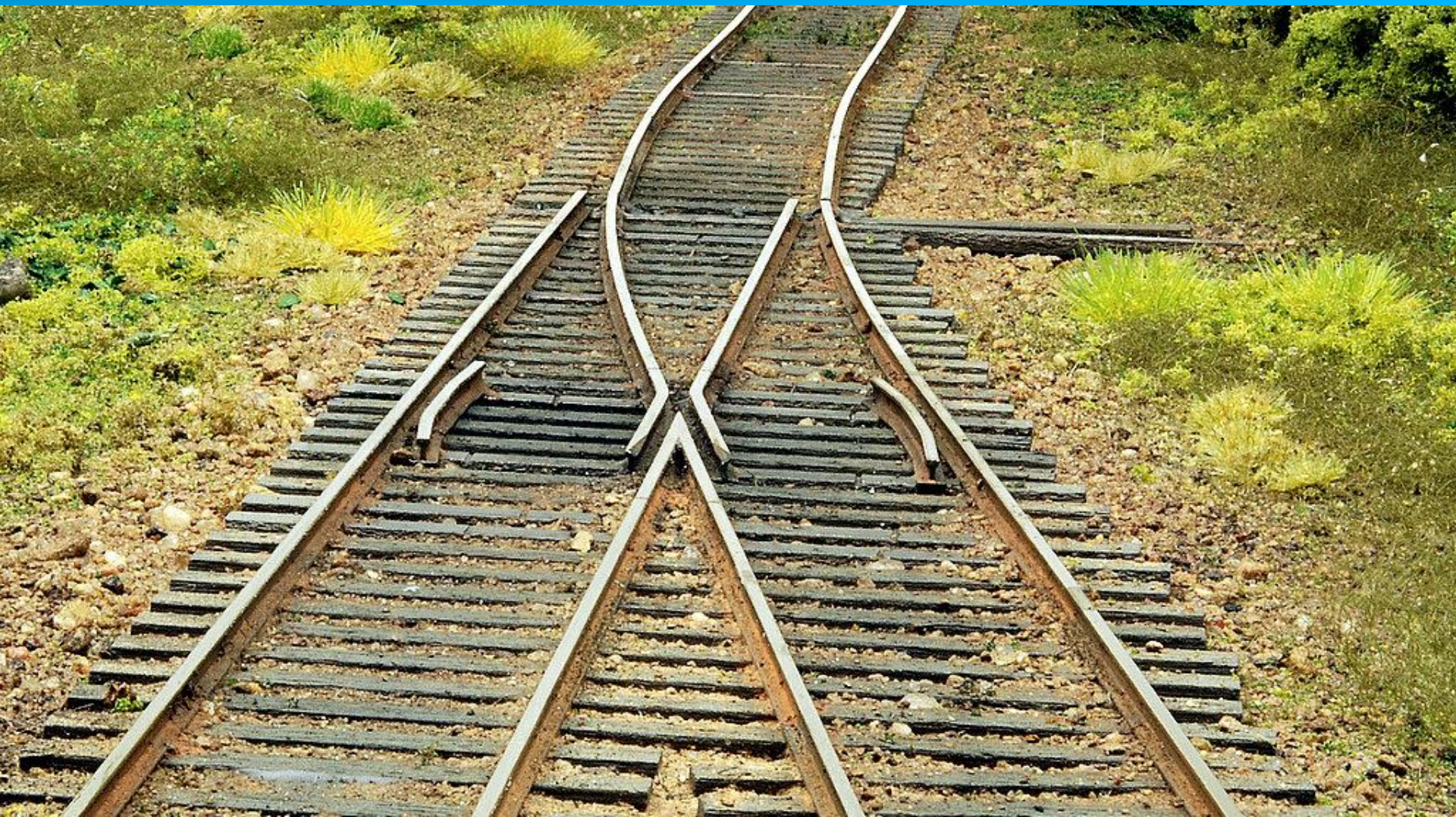
Condições de Disputa

Problemas Extras:

- São erros dinâmicos
- Não aparecem no código fonte
- Só se manifestam durante a execução
- Só quando os entrelaçamentos ocorrem
- Pode permanecer latente durante anos
- Depuração de programas contendo condições de disputa é muito complexa

Importante $\Rightarrow$ Técnicas que evitem a ocorrência

Seções Críticas



Seções Críticas

Os trechos de código de cada tarefa que acessam recursos compartilhados são denominados **seções críticas.**

Seções Críticas

Exemplo: Operação bancária

```
typedef struct conta_t
{
        int saldo ; // saldo atual da conta
        ... // outras informações da conta
} conta_t ;

void depositar (conta_t* conta, int valor)
{
        conta->saldo += valor ;
}
```

Seções Críticas

Exemplo: Operação bancária

```
typedef struct conta_t
{
        int saldo ; // saldo atual da conta
        ... // outras informações da conta
} conta_t ;

void depositar (conta_t* conta, int valor)
{
        conta->saldo += valor ;
}
```

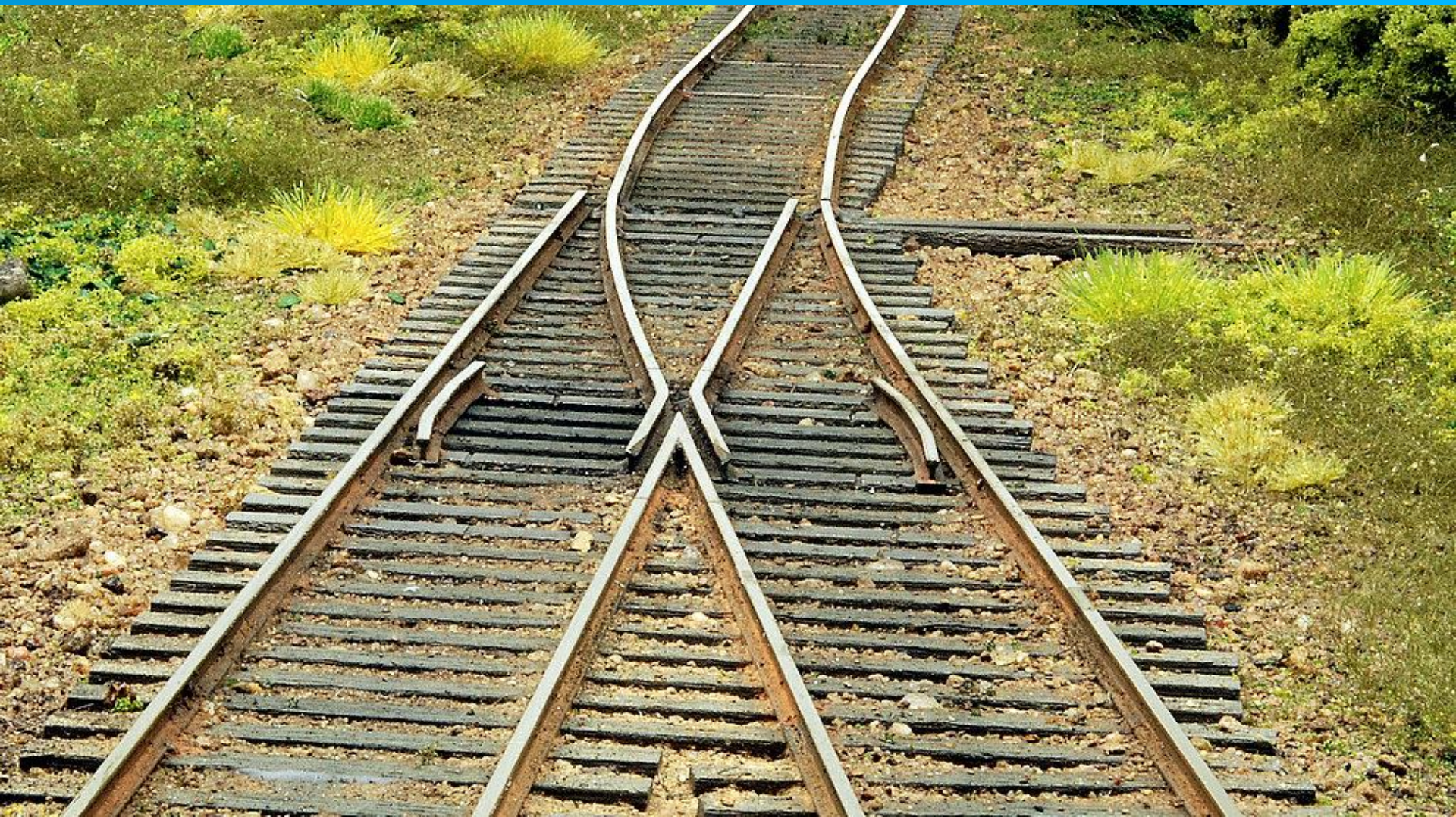
Exclusão Mútua

Evitando Problemas:

Para assegurar a operação correta de uma implementação deve-se impedir o entrelaçamento de seções críticas:

Apenas **uma** tarefa pode estar na seção crítica a cada instante. Essa propriedade é conhecida como **exclusão mútua**.

Exclusão Mútua



Mecanismos para impedir condições de disputa

- Definição do início e do fim da seção crítica
- Seção crítica $i \rightarrow cs_i$
- Primitivas de entrada e saída da seção crítica:

enter(t_a,cs_i) e *leave*(t_a,cs_i)

- Primitiva *enter* é bloqueante

```
enter();
  seção crítica;
leave();
```

Mecanismos para impedir condições de disputa

- Definição do início e do fim da seção crítica
- Seção crítica $i \rightarrow cs_i$
- Primitivas de entrada e saída da seção crítica:

enter(t_a,cs_i) e *leave*(t_a,cs_i)

- **Primitiva *enter* é bloqueante**

```
enter();
  seção crítica;
leave();
```

Definição de Mecanismos

Caso uma tarefa já esteja ocupando a seção crítica cs_i, as demais tarefas que tentarem entrar deverão aguardar até que a primeira libere a seção crítica, por meio da primitiva *leave*(cs_i)

- *enter*(t_a, cs_i) → tarefa t_a indica que deseja entrar na seção crítica cs_i

- *leave*(t_a, cs_i) → tarefa t_a informa que está saindo da seção crítica cs_i

Definição de Mecanismos

Exemplo: Operação bancária

```
typedef struct conta_t
{
        int saldo ;       // saldo atual da conta
        int numero ;      // identificação da conta (seção crítica)
        ...               // outras informações da conta
} conta_t ;

void depositar (conta_t* conta, int valor)
{
        enter (conta->numero) ; // tenta entrar na seção crítica
        conta->saldo += valor ; // está na seção crítica
        leave (conta->numero) ; // sai da seção crítica
}
```

Definição de Mecanismos

•Serão estudadas várias soluções para a implementação das primitivas *enter* e *leave*.

•As soluções propostas devem atender a alguns critérios básicos que são enumerados a seguir:

Definição de Mecanismos

•Serão estudadas várias soluções para a implementação das primitivas *enter* e *leave*.

•As soluções propostas devem atender a alguns critérios básicos que são enumerados a seguir:

Exclusão mútua : só **uma** tarefa pode estar dentro da seção crítica em cada instante.

Espera limitada : tarefa que aguarda acesso a uma seção crítica deve ter acesso garantido em um tempo **finito**.

Definição de Mecanismos

Independência de outras tarefas : decisão sobre o uso de seção crítica depende somente das tarefas que estão tentando entrar nela (*e de mais ninguém*).

Independência de fatores físicos : solução deve ser puramente lógica e não depender de fatores físicos:

- Velocidade de execução das tarefas;
- Temporizações;
- Número de processadores no sistema.

Soluções Clássicas

1 - Inibição de Interrupções

- Solução simples para implementação de *enter* e *leave*
- Troca de contexto dentro da seção crítica é desabilitada

Desvantagens:

- Risco de travamento: falta de preempção
- Dispositivos de E/S deixam de ser atendidos
- A tarefa não pode realizar operação de E/S

1 - Inibição de Interrupções

Ainda assim:

•Só funciona em sistemas mono-processados;

•Em máquinas multiprocessadas, duas tarefas concorrentes podem executar simultaneamente em processadores separados.

- Lembre que a inibição das interrupções é feita por processador

2 – Espera Ocupada

Consiste em testar continuamente uma condição que indica se a seção desejada está livre ou ocupada.

2.1 – Solução mais simples

```
int busy = 0 ;              // a seção está inicialmente livre
                            // variável global

void enter (int task)
{
        while (busy) ; // espera enquanto a seção estiver ocupada
        busy = 1 ;      // marca a seção como ocupada
}

void leave (int task)
{
        busy = 0 ;      // libera a seção (marca como livre)
}
```

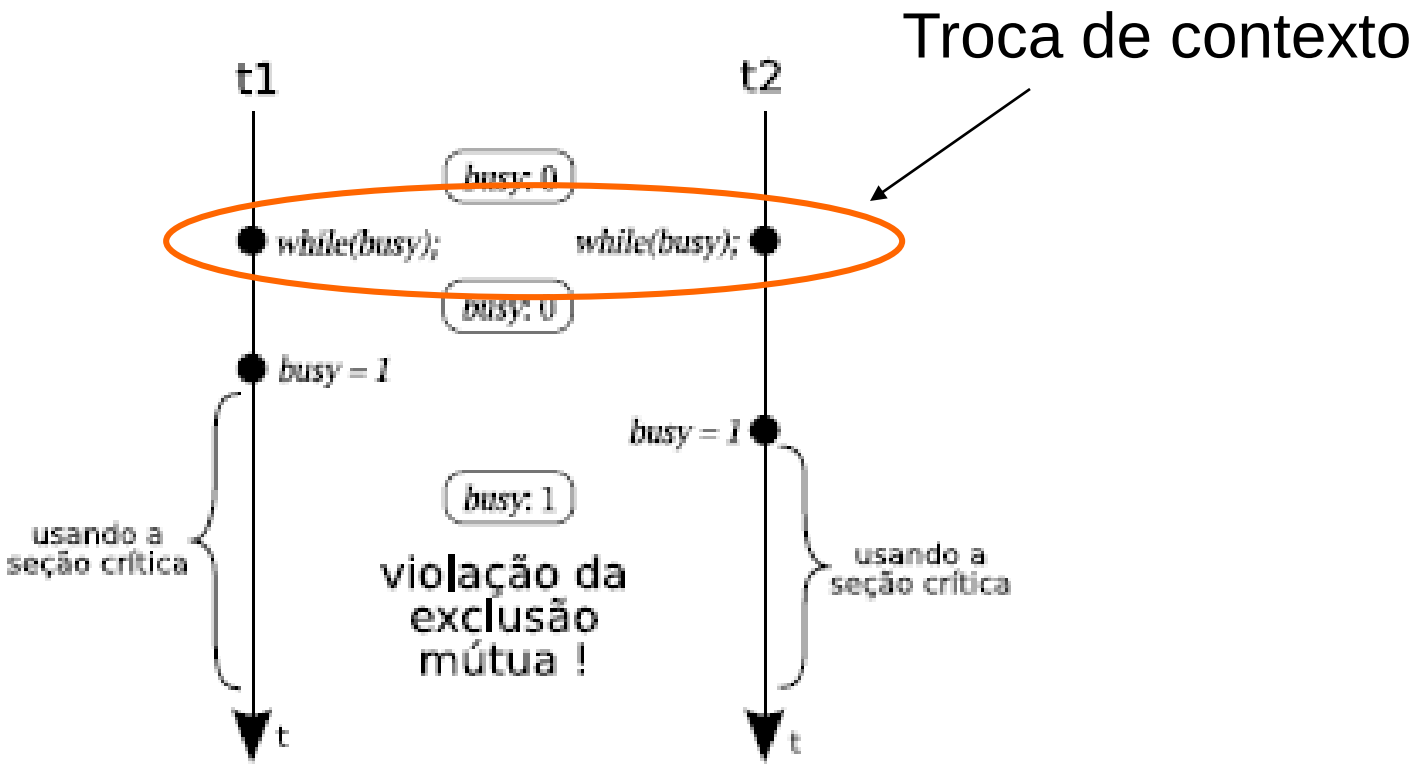
2.1 – Solução mais simples

```
int busy = 0 ;              // a seção está inicialmente livre
                            // variável global


void enter (int task)
{
        while (busy) ; // espera enquanto a seção estiver ocupada
        busy = 1 ;      // marca a seção como ocupada
}

void leave (int task)
{
        busy = 0 ;      // libera a seção (marca como livre)
}
```

NÃO FUNCIONA

2.1 – Solução mais simples

O teste da variável *busy* e sua atribuição são feitos em momentos distintos.



2.2 – Alternância de Uso

```
int turn = 0 ;
int num_tasks ;

void enter (int task)                  // task vale 0, 1, ..., num_tasks-1
{
        while (turn != task) ;   // a tarefa espera sua vez
}

void leave (int task)
{
        if (turn < num_tasks-1) // a vez é da próxima tarefa
                turn ++ ;
        else
                turn = 0 ;          // volta à primeira tarefa
}
```

2.2 – Alternância de Uso

Problema:

Cada tarefa aguarda seu turno em uma sequência circular: $t0 \rightarrow t1 \rightarrow t2 \rightarrow \cdots \rightarrow tn-1 \rightarrow t0$. ✓

Garante exclusão mútua e independe de fatores externos ✓

Não atende ao critério “espera limitada”: caso uma tarefa t_i não deseje usar a seção crítica, todas as tarefas t_j com $j > i$ ficarão impedidas de fazê-lo. X

2.3 – Solução de Peterson

- Solução correta para a exclusão mútua
- Primeira proposição: T. Dekker em 1965
  - http://www.cs.utexas.edu/users/EWD/transcriptions/EWD01xx/EWD123.html
- Em 1981 Gary Peterson propôs uma solução mais simples

2.3 – Solução de Peterson (2 tarefas)

```
int turn = 0 ;                          // indica de quem é a vez
int wants[2] = {0, 0} ;                 // indica se a tarefa i quer acessar a
                                        // seção crítica

void enter (int task)                   // task pode valer 0 ou 1
{
        int other = 1 - task ;          // indica a outra tarefa
        wants[task] = 1 ;               // task quer acessar a seção crítica
        turn = task;
        while ((turn = = task) && wants[other]) ; // espera ocupada
}
void leave (int task)
{
        wants[task] = 0 ;               // task libera a seção crítica
}
```

2.3 – Solução de Peterson (2 tarefas)

Tarefa 0

```
void enter (int 0)
{
        wants[0] = 1 ;
        turn = 0;
        while ((turn = = 0) && wants[1]) ;

.....

}
void leave (int 0)
{
        wants[0] = 0 ;
}
```

Tarefa 1

```
void enter (int 1)
{
        wants[1] = 1 ;
        turn = 1;
        while ((turn = = 1) && wants[0]) ;

.....

}
void leave (int 1)
{
        wants[1] = 0 ;
}
```

2.4 – Instruções *Test-and-Set*

```
int busy = 0 ;              // a seção está inicialmente livre

void enter (int task)
{
        while (busy) ; // espera enquanto a seção estiver ocupada
        busy = 1 ;      // marca a seção como ocupada
}

void leave (int task)
{
        busy = 0 ;      // libera a seção (marca como livre)
}
```

NÃO FUNCIONA

2.4 – Instruções *Test-and-Set*

Solução $\Rightarrow$ teste e atribuição de valor a uma variável de forma *atômica*.

$\rightarrow$ *Não há troca de contexto entre as 2 operações.*

2.4 – Instruções *Test-and-Set*

Exemplo: *Test-and-Set Lock* (TSL)

Pseudocódigo:

$TSL(x)$: $old \leftarrow x$

$x \leftarrow 1$

$return(old)$

Uso → while(TSL(busy))

{

.....

}

2.4 – Instruções *Test-and-Set*

Implementação das primitivas *enter* e *leave* usando a instrução TSL:

```
int busy = 0 ;                          // variável de trava

void enter (int busy)                   // passa a trava
{
        while ( TSL (busy) ) ;          // espera ocupada
}

void leave (int busy)
{
        (busy) = 0 ;                    // libera a seção crítica
}
```

2.4 – Instruções *Test-and-Set*

TSL são amplamente usados no interior do sistema operacional para controlar o acesso a seções críticas internas do núcleo:

- descritores de tarefas;
- buffers de arquivos;
- ou de conexões de rede;
- etc.

Porém, soluções de espera ocupada são inadequadas para a construção de aplicações de usuário.

Espera Ocupada - Problemas

Ineficiência :

•Tarefas que aguardam o acesso a uma seção crítica ficam testando continuamente uma condição, consumindo tempo de processador sem necessidade.

•O procedimento adequado seria suspender essas tarefas até que a seção crítica solicitada seja liberada.

Espera Ocupada - Problemas

```
void enter (int lock)          // passa a trava
{
        while ( TSL (lock) ) ; // espera ocupada
}
```

Espera Ocupada - Problemas

Injustiça :

•Não há garantia da ordem no acesso à seção crítica;

•Dependendo da duração de *quantum* e da política de escalonamento, uma tarefa pode entrar e sair da seção crítica várias vezes, antes de outras tarefas.

Espera Ocupada - Problemas

Soluções com espera ocupada são pouco usadas na construção de aplicações. Seu maior uso se encontra na programação de estruturas de controle de concorrência dentro do núcleo do sistema operacional.