

# DCA-0125 Sistemas de Tempo Real

Luiz Affonso Guedes  
[www.dca.ufrn.br/~affonso](http://www.dca.ufrn.br/~affonso)  
[affonso@dca.ufrn.br](mailto:affonso@dca.ufrn.br)



# Introdução à Programação Concorrente

# Objetivos

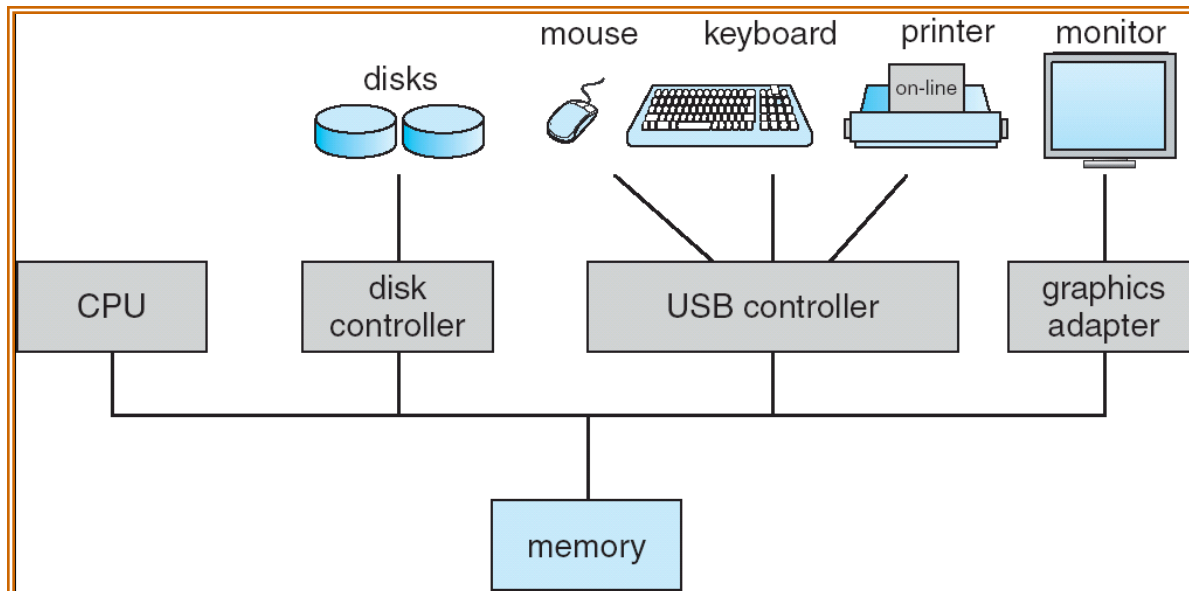
- ❑ Apresentar os principais conceitos e paradigmas associados com programação concorrente.
- ❑ Associar os paradigmas e problemas de programação concorrente com o escopo dos Sistemas Operacionais

# Conteúdo

- ❑ Caracterização e escopo da programação concorrente.
- ❑ Abstrações e Paradigmas em Programação Concorrente
  - Tarefas, região crítica, sincronização, comunicação.
- ❑ Propriedades de sistemas concorrentes
  - Exclusão mútua, Starvation e DeadLock
- ❑ Primitivas de Programação Concorrente
  - Mutex, Semáforos, monitores
  - Memória compartilhada e troca de mensagens
- ❑ Problemas clássicos em programação concorrente
  - Produtor-consumidor
  - Leitores e escritores
  - Jantar dos filósofos

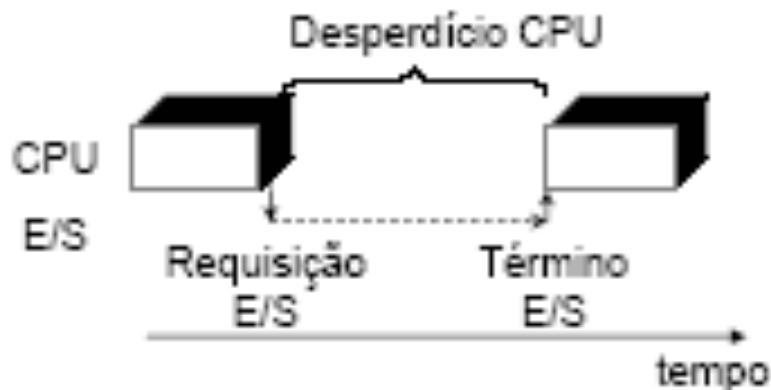
# Recordando

- ❑ Cenário Atual dos Sistemas Operacionais
  - Uma ou mais CPUs, controladores de devices conectados via uma barramento comum, acessando memórias compartilhadas.
  - Execução **concorrente** de CPUs e devices competindo por recursos.



# Recordando

- ❑ Para se construir SO eficientes, há a necessidade Multiprogramação!



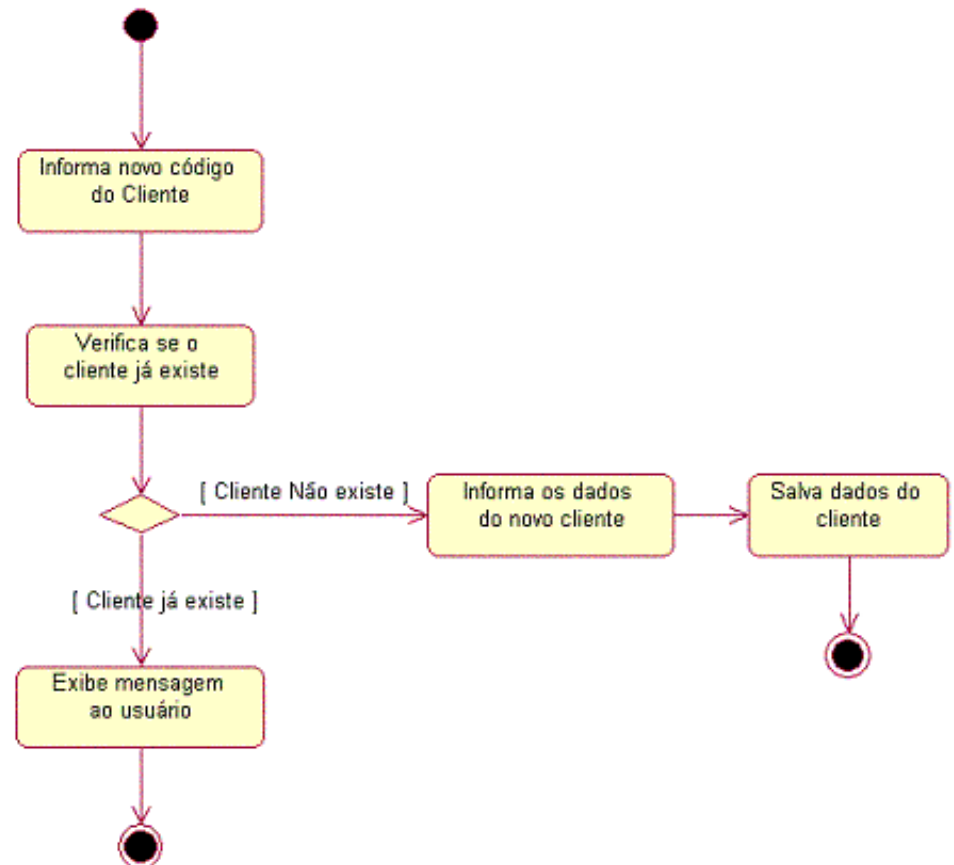
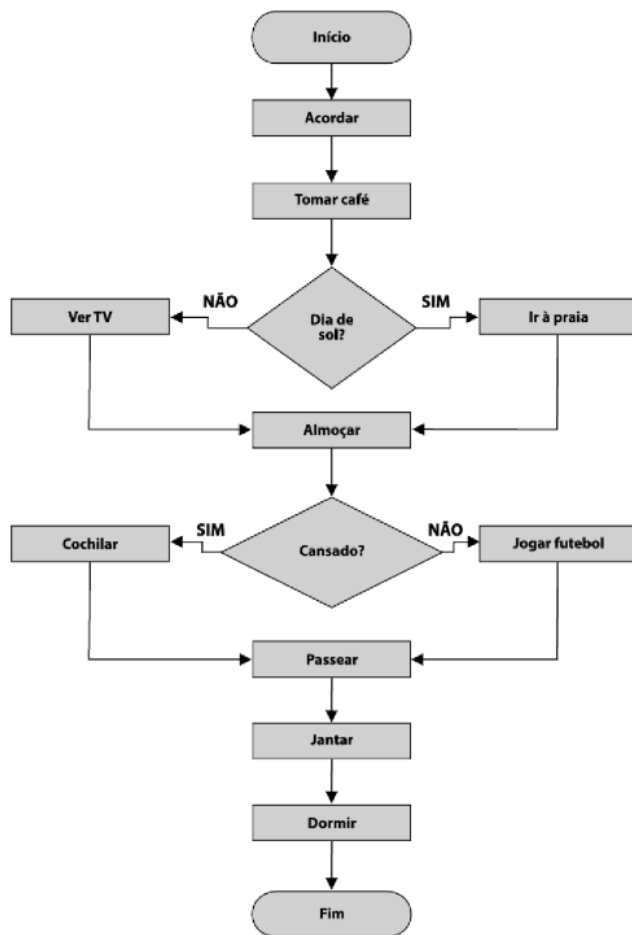
# Consequências da Multiprogramação

- ❑ Necessidade de controle e sincronização dos diversos programas.
- ❑ Necessidade de se criar conceitos e abstração novos
  - Modelagem
  - Implementação
- ❑ Necessidade de se estudar os paradigmas da Programação Concorrente!

# Conceitos de Programação

## ❑ Programação Sequencial:

- Programa com apenas um fluxo de execução.

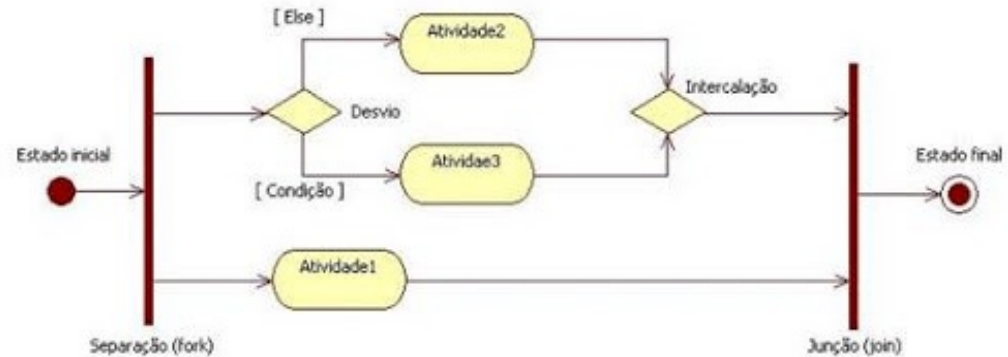




# Conceitos de Programação

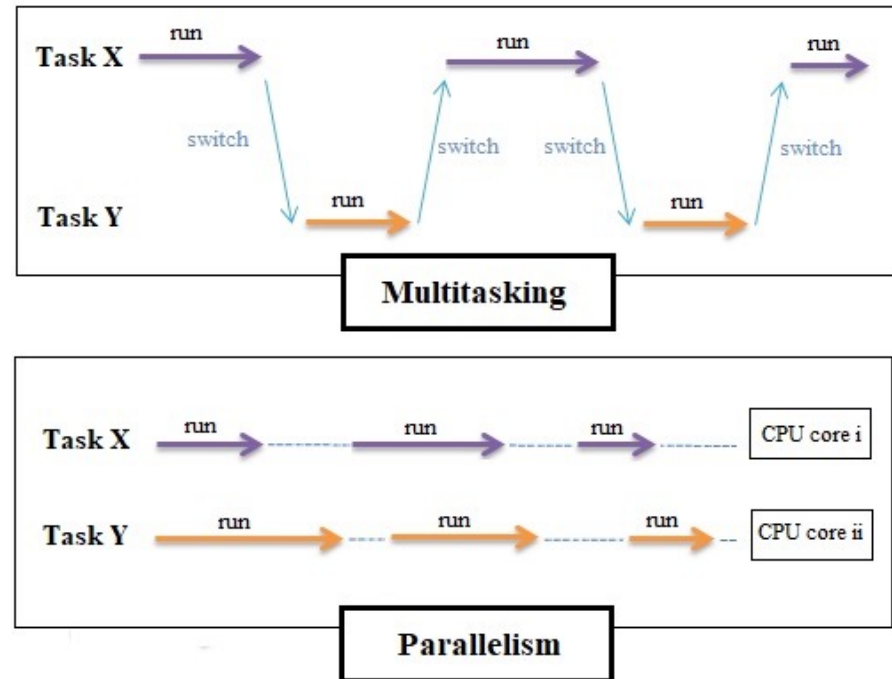
## □ Programação Concorrente:

- Possui dois ou mais fluxos de execução sequenciais, que podem ser executados concorrentemente no tempo.
- Necessidade de comunicação para troca de informação e sincronização.
  - Aumento da eficiência e da complexidade



# Paralelismo X Concorrência

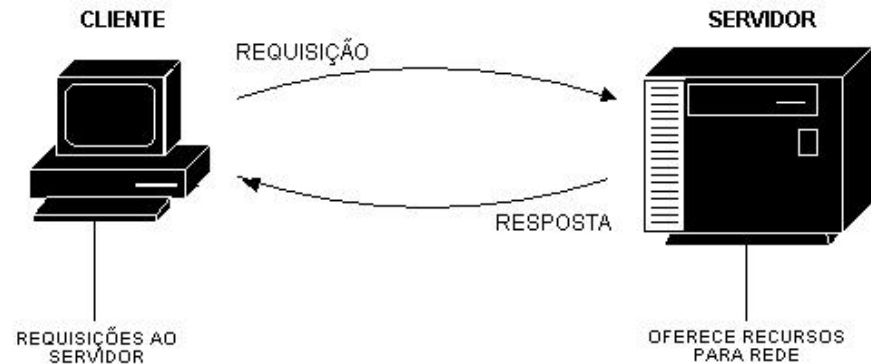
- ❑ Paralelismo real só ocorre em máquinas multiprocessadas.
- ❑ Paralelismo aparente (concorrência) é um mecanismo de se executar "simultaneamente"  $M$  programas em  $N$  Processadores, quando  $M > N$ .
  - $N = 1$ , caso particular de monoprocessamento.



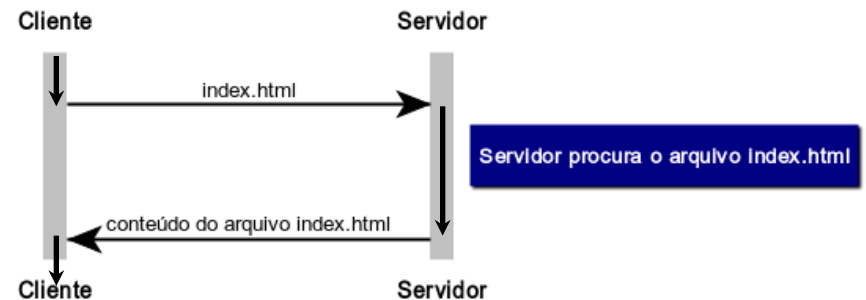
# Paralelismo X Concorrência

- ❑ Programação Distribuída é programação concorrente ou paralela????

- Modelo cliente-servidor?



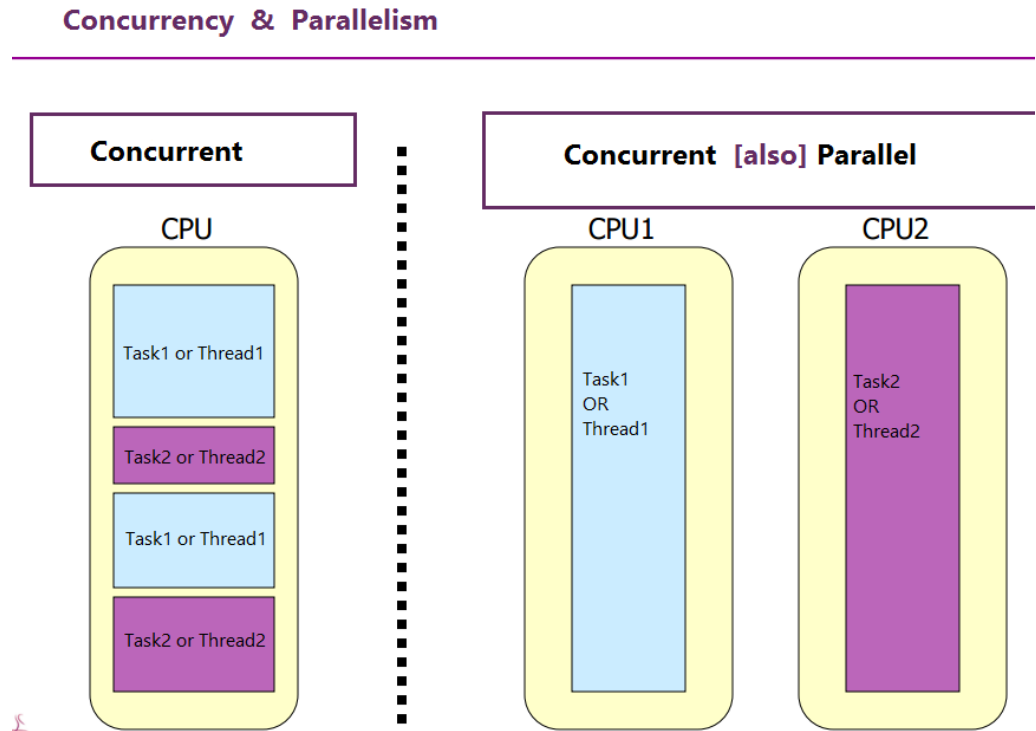
Comunicação cliente-servidor na internet



[www.websequencediagrams.com](http://www.websequencediagrams.com)

# Paralelismo X Concorrência

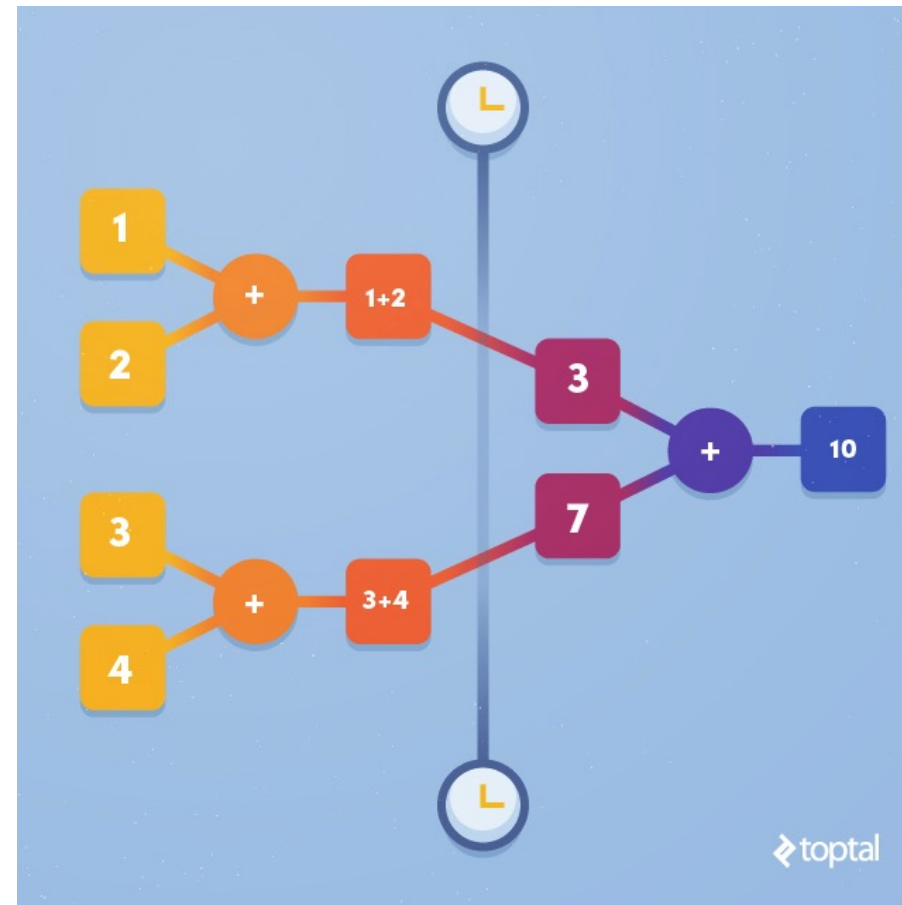
- ❑ Programação Distribuída é programação concorrente ou paralela????



# Programação Concorrente

- ❑ Paradigma de programação que possibilita a implementação computacional de vários programas sequenciais, que executam "**simultaneamente**" trocando informações e disputando recursos comuns.

$$(1+2) + (3+4) = X$$
$$X1 + X2 = X$$



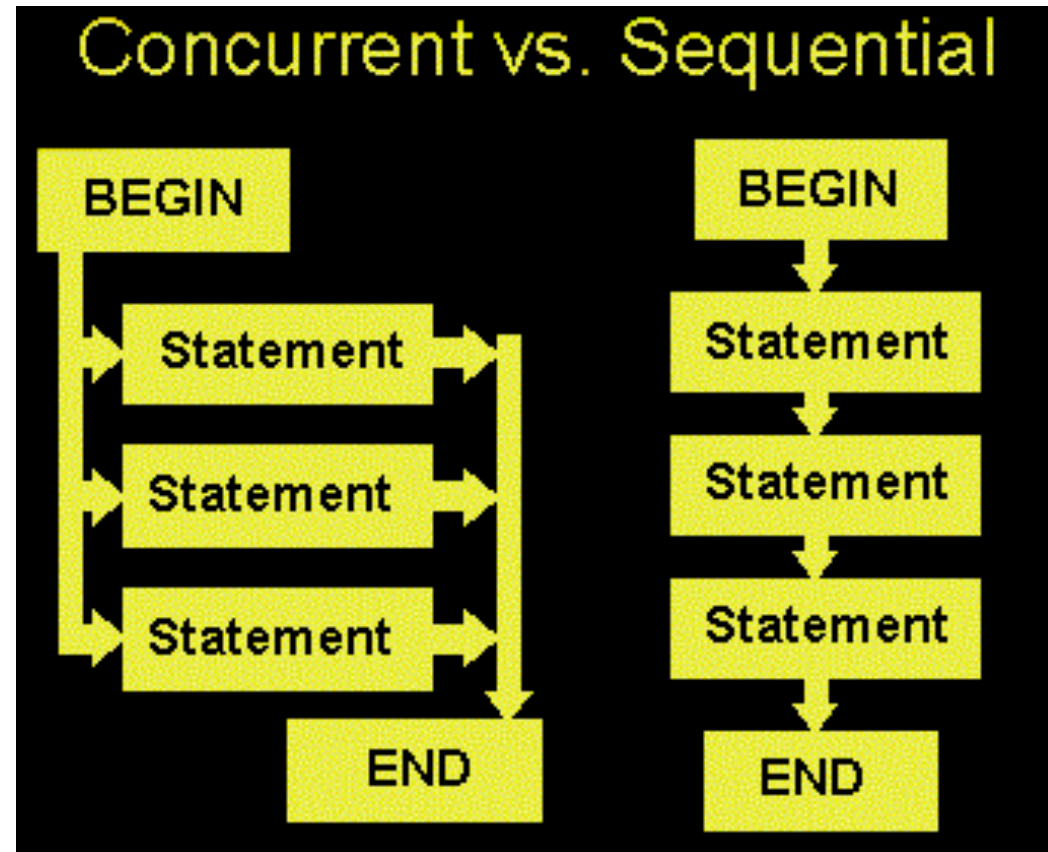
# Programação Concorrente

- Programas cooperantes
  - Quando podem afetar ou ser afetados entre si.



# Motivação para o Uso da Programação Concorrente

- ❑ Aumento do desempenho
  - Possibilidade de se implementar multiprogramação.
- ❑ Possibilidade de desenvolvimento de aplicações que possuem paralelismo intrínseco.
  - SO Modernos, por exemplo.
  - Sistema de Automação Industrial.





# Desvantagens da Programação Concorrente

## ❑ Programas mais complexos

- Pois há, agora, vários fluxos de programas sendo executados concorrentemente.
- Esses fluxos podem interferir uns nos outros.



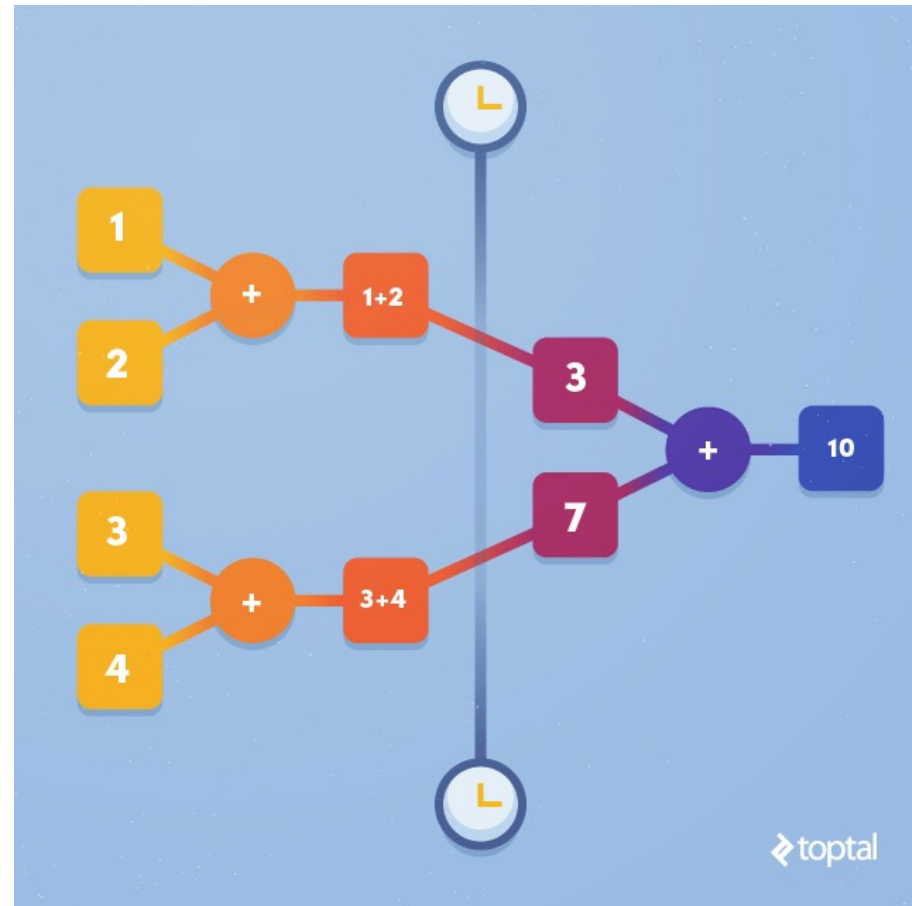


# Desvantagens da Programação Concorrente

- ❑ Execução não determinística
  - Na programação sequencial, para um dado conjunto de entrada, o programa irá apresentar o mesmo conjunto de saída.
  - Em programação concorrente, isto não é necessariamente verdade.

$$(1+2) + (3+4) = X$$
$$X1 + X2 = X$$

$X1 = 0; X2=0; X = 0;$   
 $X1 = (1+2);$   
 $X2 = (3+4);$   
 $X = X1 + X2;$



# Resumindo ...

## ○ Programação Concorrente

- As questões básicas estão associadas com a necessidade de **comunicação** e **sincronização** entre as tarefas.

# Resumindo ...

## ❑ Programação Concorrente X Programação Sequencial

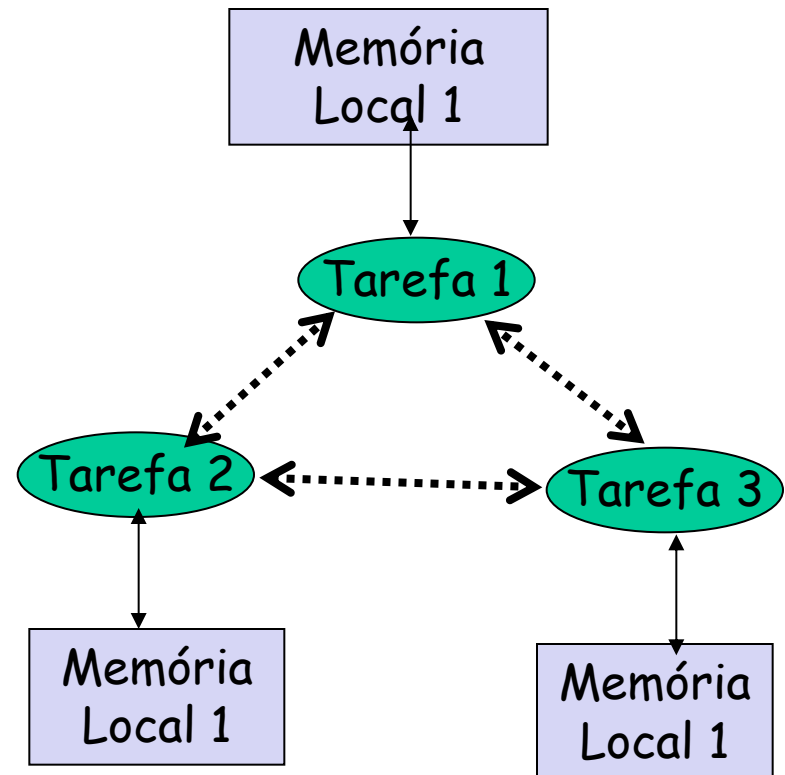
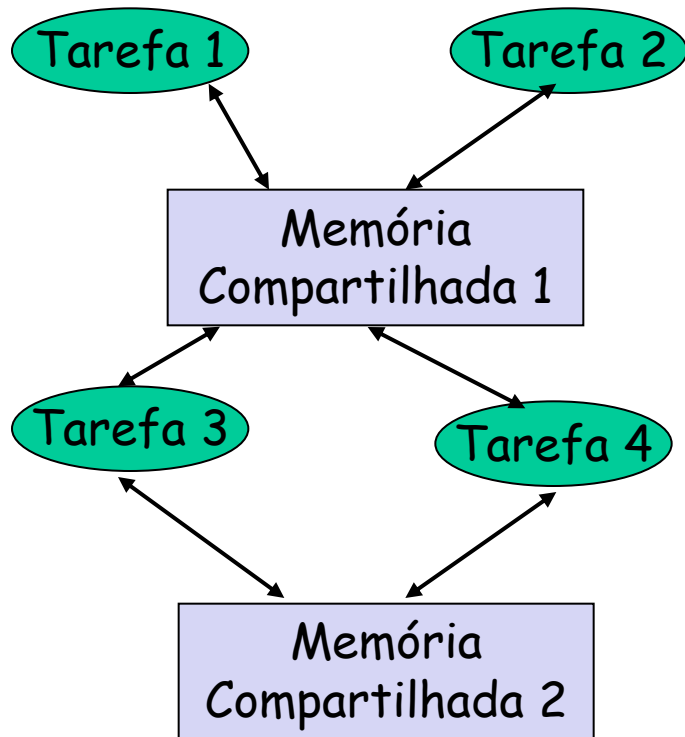
- A programação sequencial possui a característica de produzir o mesmo resultado para o mesmo conjunto de dados de entrada.
- A programação concorrente possibilita que várias atividades, que constituem o programa, sejam executadas superpostas no tempo.
- A programação concorrente é composta de vários programas sequenciais sendo executados de forma concorrente.

# Elementos Básicos da Programação Concorrente

- ❑ Tarefas (Processos ou Threads)
- ❑ Sincronismo
- ❑ Troca de Informação

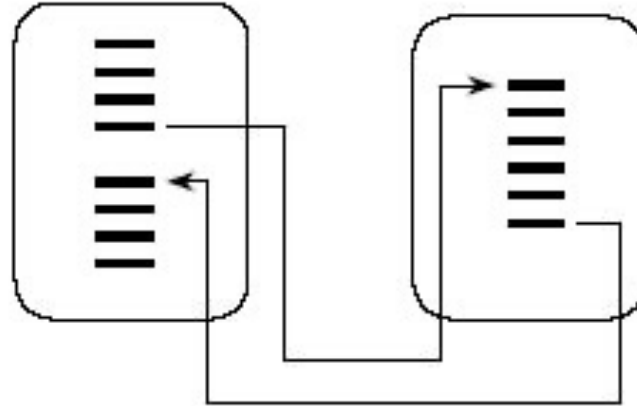
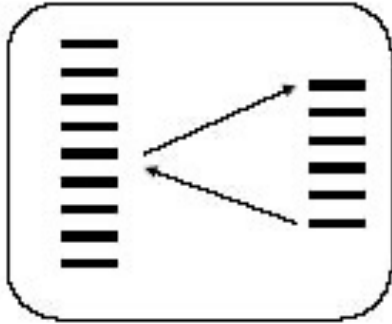
# Paradigmas para Troca de Informação

- ❑ Troca de Informação: **comunicação por memória compartilhada** e **comunicação via troca de mensagem**



# Paradigmas para Troca de Informação

- ❑ Remote Procedure Call (RPC)
  - Primitiva baseada no paradigma de linguagens procedurais.

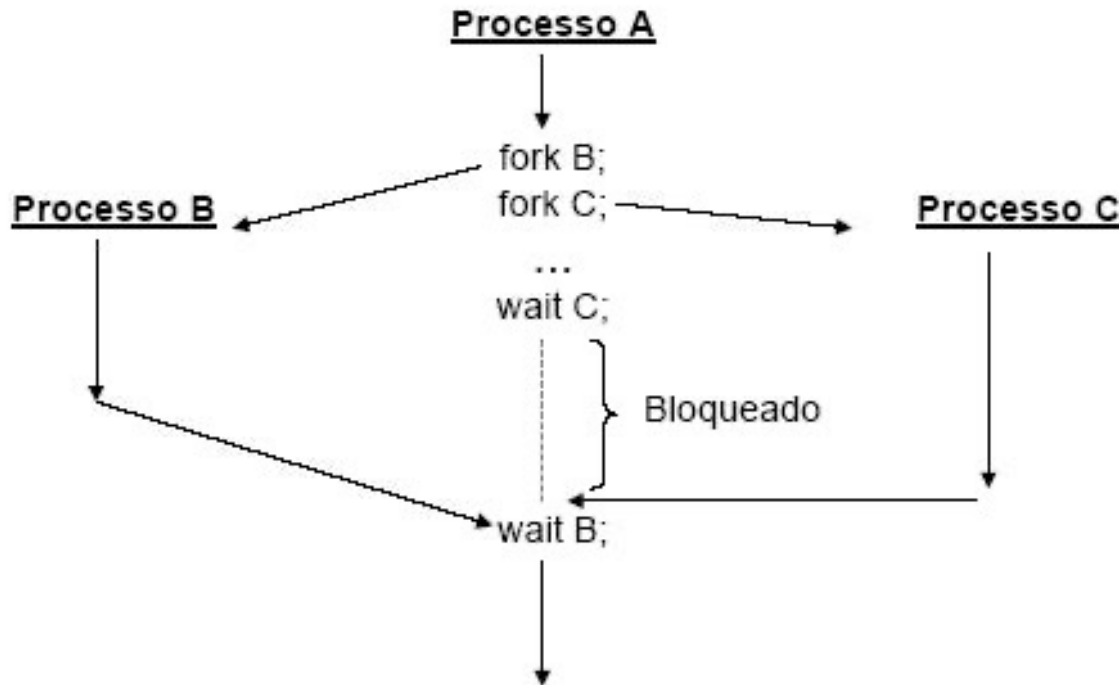


# Especificação das Tarefas

- ❑ Quantas tarefas concorrentes haverá no sistema?
- ❑ O quê cada tarefa uma fará?
- ❑ Como as tarefas irão cooperar entre si?
  - Comunicação entre tarefas.
- ❑ Quais recursos elas irão disputar?
  - Necessidade de mecanismo de controle de acesso a recursos.
  - Memória, cpu, devices, etc.
- ❑ Qual é a ordem que as tarefas devem executar?
  - Sincronismo entre tarefas.

# Primitivas de Paralelismo

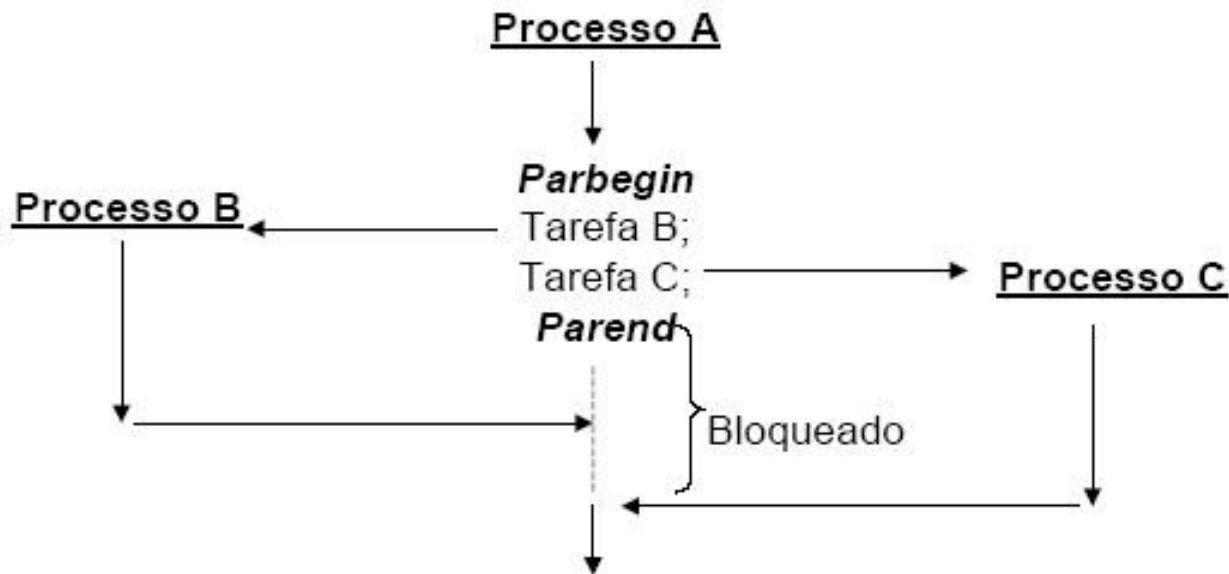
## ❑ fork/joint (fork/wait)





# Primitivas de Paralelismo

## □ parbegin/parend



# Problema do Compartilhamento de Recursos

- ❑ Programação concorrente implica em compartilhamento de recursos
- ❑ Como manter o estado (dados) de cada tarefa (fluxo) coerente e correto mesmo quando diversos fluxos se interagem?
- ❑ Como garantir o acesso a um determinado recurso a todas as tarefas que necessitarem dele?
  - Uso de CPU, por exemplo.

# Problema de Condição de Corrida

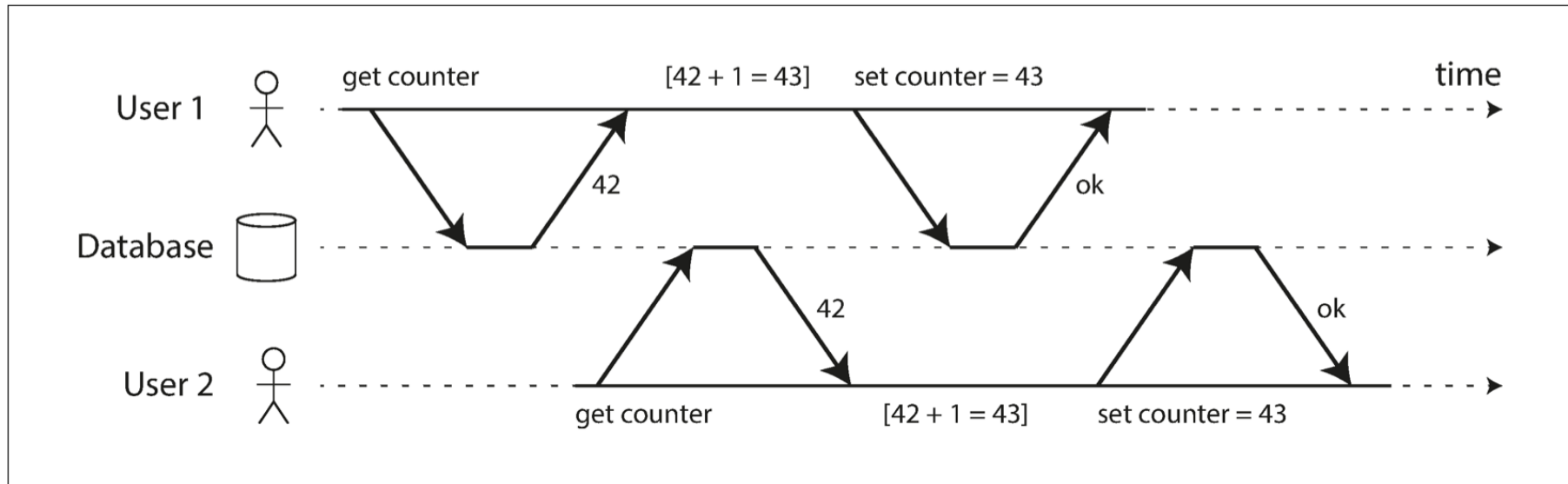
- ❑ Ocorre quando duas ou mais tarefas manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os acessos são efetuados.
  - Há a necessidade de um mecanismo de controle, senão o resultado pode ser imprevisível.



Thread 1	Thread 2		Inteiro
			0
lê o valor		←	0
incrementa-o			0
escreve nele		→	1
	lê o valor	←	1
	incrementa-o		1
	escreve nele	→	2

# Problema de Condição de Corrida

- ❑ Ocorre quando duas ou mais tarefas manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os acessos são efetuados.
  - Há a necessidade de um mecanismo de controle, senão o resultado pode ser imprevisível.



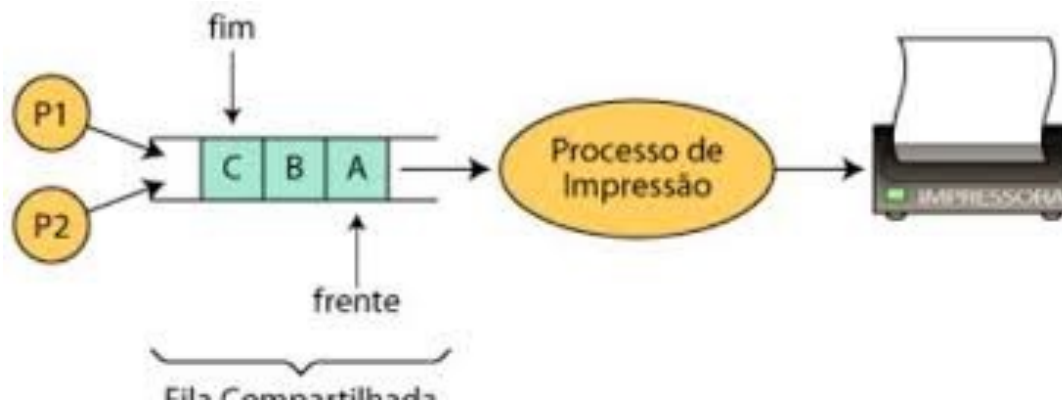
*A race condition between two clients concurrently incrementing a counter.*

# Requisitos Básicos

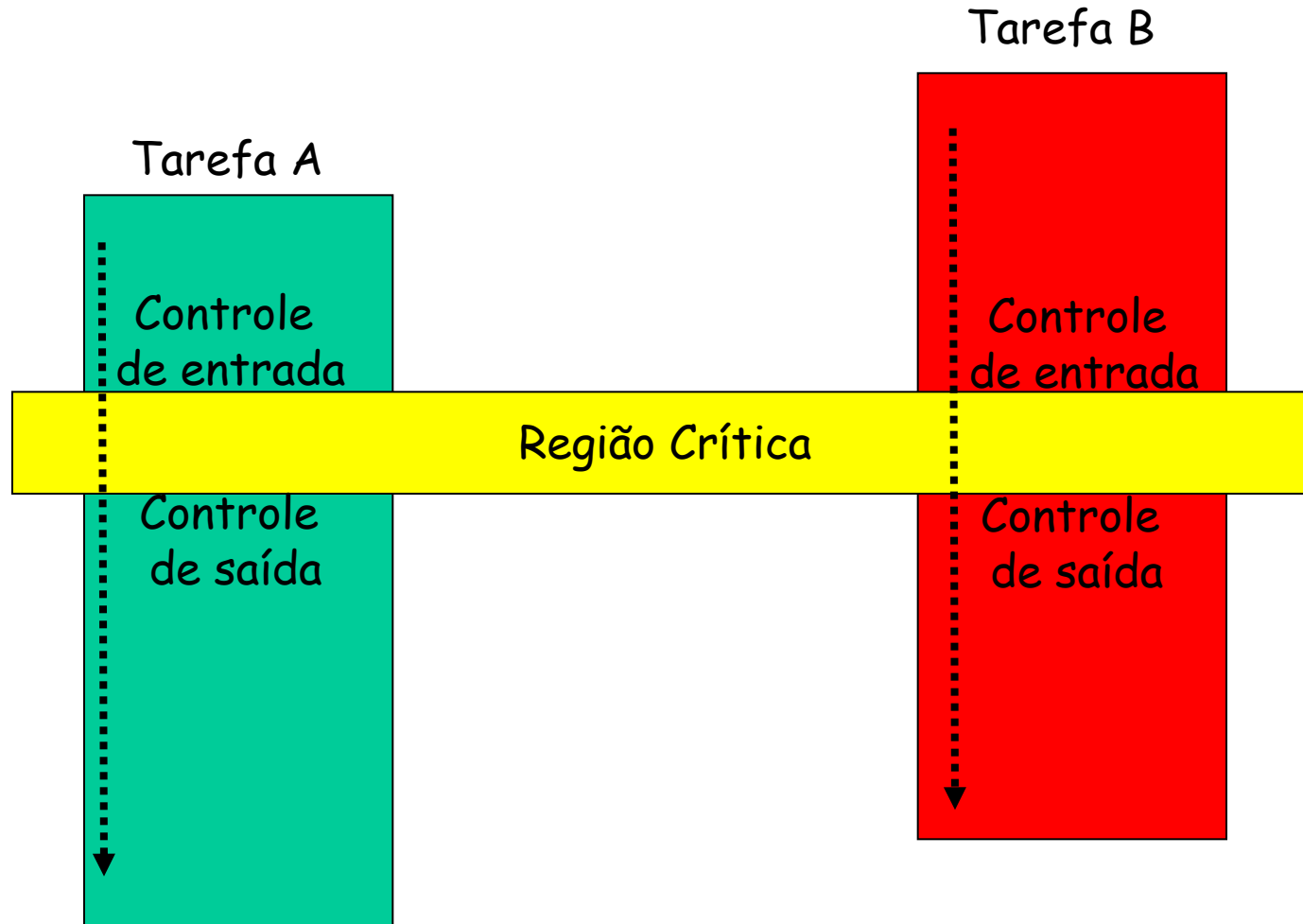
- ❑ Eliminar problemas de corridas
- ❑ Criar um protocolo que as diversas tarefas possam cooperar sem afetar a consistência dos dados
- ❑ Controle de acesso a regiões críticas
  - Implementação de mecanismos de exclusão mútua.

# Problema de Exclusão Mútua

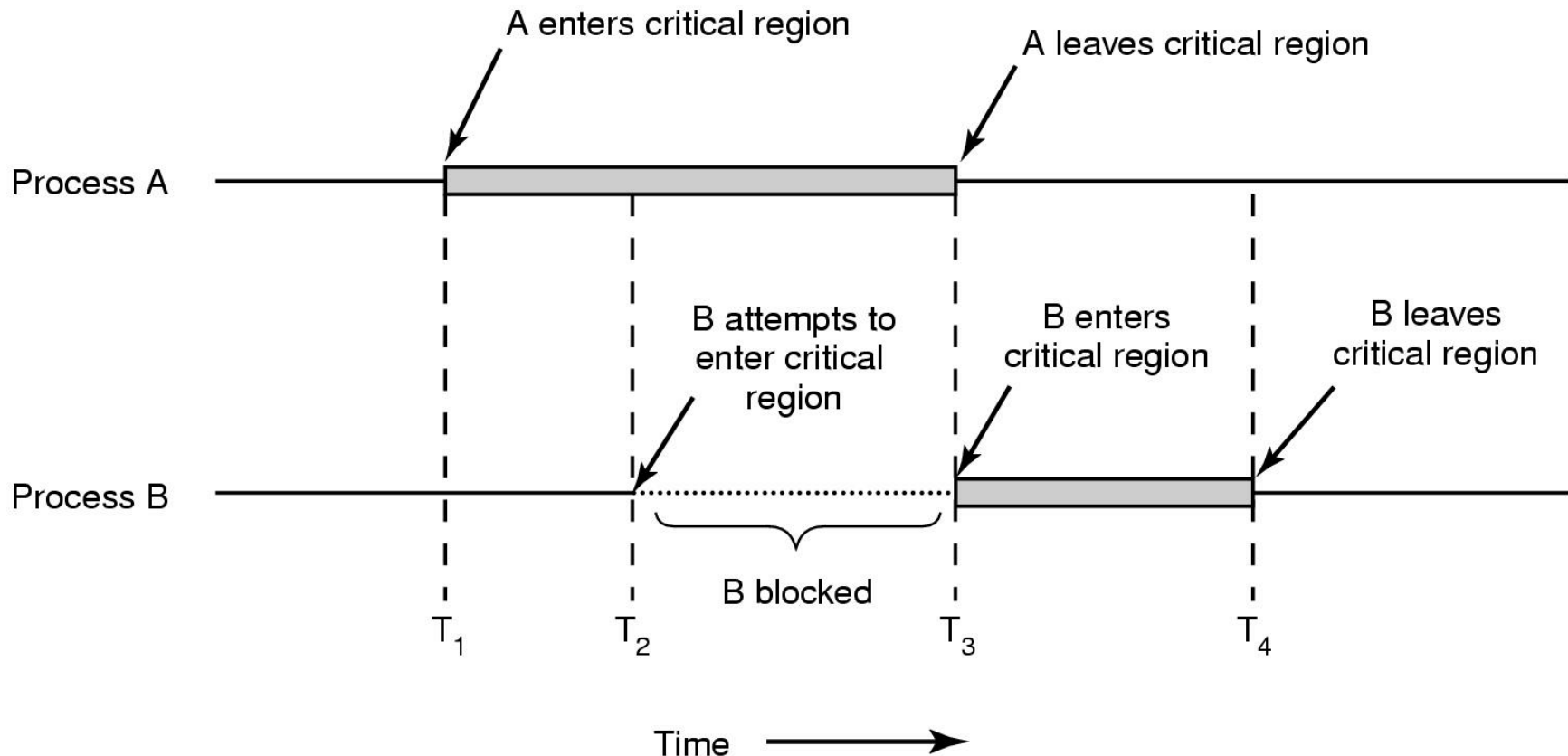
- ❑ Ocorre quando duas ou mais tarefas necessitam de recursos de forma exclusiva:
  - CPU, impressora, dados, etc.
  - → esse recurso é modelado como uma região crítica.



# Exclusão Mútua - Idéia Básica



# Exclusão Mútua - Idéia Básica





# Propriedades da Região Crítica

- ❑ Regra 1 - Exclusão Mútua
  - Duas ou mais tarefas não podem estar simultaneamente numa mesma região crítica.
- ❑ Regra 2 - Progressão
  - Nenhuma tarefa fora da região crítica pode bloquear a execução de uma outra tarefa.
- ❑ Regra 3 - Espera Limitada (Starvation)
  - Nenhuma tarefa deve esperar infinitamente para entrar em uma região crítica.
- ❑ Regra 4 -
  - Não fazer considerações sobre o número de processadores e nem sobre suas velocidades relativas (condição de corrida).

# Implementação Iniciais de Exclusão Mútua

- ❑ Desativação de Interrupção
- ❑ Uso de variáveis especiais de **lock**
- ❑ Alternância de execução

# Desabilitação de Interrupção

- ❑ Não há troca de tarefas com a ocorrência de interrupções de tempo ou de eventos externos.
- ❑ Desvantagens:
  - Uma tarefa pode dominar os recursos.
  - Não funciona em máquinas multiprocessadas, pois apenas a CPU que realiza a instrução é afetada (violação da regra 4)

Tarefa 1



# Variável do Tipo Lock

- ❑ Criação de uma variável especial compartilhada que pode assumir dois valores:
  - Zero → livre
  - 1 → ocupado
- ❑ Desvantagem:
  - Apresenta condição de corrida.

```
.  
.   
.   
while (lock == 1);  
lock = 1;  
Regiao_critica();  
lock = 0;  
.   
.   
.
```

Espera ocupada

Se houver uma  
interrupção nesse ponto?

# Alternância

- ❑ As tarefas se intercalam no acesso da região crítica.
- ❑ Desvantagem
  - Teste contínuo do valor da variável
    - Desperdício do processador: **espera ocupada**
  - Se as tarefas não necessitarem utilizar a região crítica com a mesma frequência.

Tarefa 1

```
while(true){  
    while(vez != 0);  
    Regiao_critica();  
    vez = 1;  
    Regiao_nao_critica();  
}
```

Tarefa 2

```
while(true){  
    while(vez != 1);  
    Regiao_critica();  
    vez = 0;  
    Regiao_nao_critica();  
}
```

# Alternância

Evitando a espera ocupada

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

void *thread_function(void *arg);
int run_now = 1;
char message[] = "Hello World";

int main() {
    int res;
    pthread_t a_thread;
    void *thread_result;
    int print_count1 = 0;

    res = pthread_create(&a_thread, NULL, thread_function, (void *)message);
    if (res != 0) {
        perror("Thread creation failed");
        exit(EXIT_FAILURE);
    }

    while(print_count1++ < 20) {
        if (run_now == 1) {
            printf("MAIN() --> 1\n");
            run_now = 2;
        }
        else {
            printf("MAIN() --> Vai dormir por 1 segundo\n");
            sleep(1);
        }
    }

    printf("\nMAIN() --> Esperando a thread terminar...\n");
    res = pthread_join(a_thread, &thread_result);
    if (res != 0) {
        perror("Thread join failed");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() --> A Thread foi terminada\n");
    exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
    int print_count2 = 0;

    while(print_count2++ < 20) {
        if (run_now == 2) {
            printf("THREAD --> 2\n");
            run_now = 1;
        }
        else {
            printf("THREAD() --> Vai dormir por 1 segundo\n");
            sleep(1);
        }
    }

    printf("THREAD() --> Vai dormir por 3 segundos\n");
    sleep(3);
    printf("THREAD() --> Vai terminar thread\n");
}
```

# Mecanismos Modernos de Exclusão Mútua

- ❑ Abordagens Iniciais - Algorítmicas
  - Combinações de variáveis do tipo lock e alternância (Dekker 1965, Peterson 1981)
  - Ineficientes
- ❑ Abordagens Modernas
  - Primitivas implementadas na linguagem e suportadas pelo SO.
    - Mais eficientes.
    - Não há espera ocupada.
  - Mutex, Semáforo e Monitores

Disciplina: Sistema de Tempo Real

# Introdução à Programação Concorrente

## Continuação




# Mutex

- ❑ Variável compartilhada para controle de acesso a região crítica.
- ❑ CPU são projetadas levando-se em conta a possibilidade do uso de múltiplos processos.
- ❑ Inclusão de duas instruções **assembly** para leitura e escrita de posições de memória de forma **atômica**.

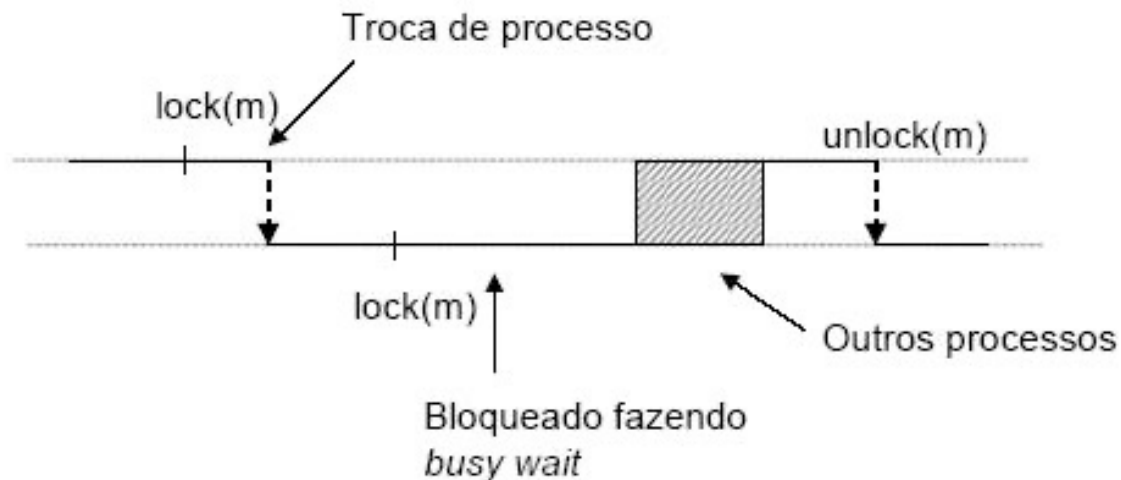
```
...  
lock(flag);  
Regiao_critica();  
unlock(flag);  
Regiao_nao_critica();  
...
```

Operações atômicas



# Mutex

- ❑ Implementação com espera ocupada (busy waiting)
  - Inversão de prioridade
  - Solução ineficiente



# Mutex

## ❑ Implementação Bloqueante

### ○ Evita a espera ocupada

- Ao acessar um flag ocupado `lock(flag)`, o processo é bloqueado.
- O processo só é desbloqueado quando o flag é desocupado.

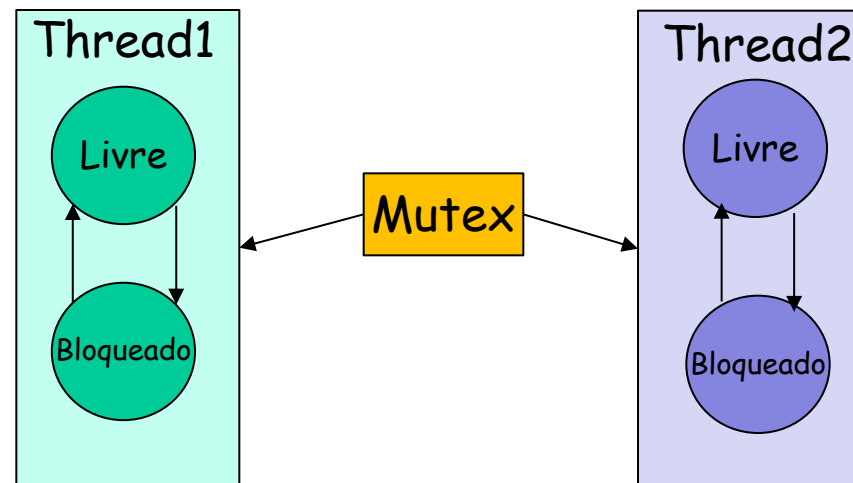
### ○ Implementação de duas primitivas

- `Sleep` → bloqueia um processo a espera de uma sinalização.
- `Wakeup` → sinaliza a liberação de um processo.

# Mutex

## ❑ Exercício: uso de mutex na Pthread

- Compile e execute o programa thread07.cpp
  - `g++ -o thread07 thread07.cpp -lpthread`
  - `./thread07`
  - Observe o código e analise o resultado quando se modifique os valores de tempo de sleeps.



```
...  
lock(flag);  
Regiao_critica();  
unlock(flag);  
Regiao_nao_critica();  
...
```

```
...  
lock(flag);  
Regiao_critica();  
unlock(flag);  
Regiao_nao_critica();  
...
```

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

void *thread_function1(void *arg);
void *thread_function2(void *arg);

pthread_mutex_t work_mutex; /* declaração de um mutex */

#define WORK_SIZE 1024
char work_area[WORK_SIZE];
int time_to_exit = 0;

int main() {
    int res;
    pthread_t thread1, thread2; // declaração de 02 threads
    void *thread_result;

    res = pthread_mutex_init(&work_mutex, NULL); // criação do mutex
    if (res != 0) {
        perror("Iniciação do Mutex falhou");
        exit(EXIT_FAILURE);
    }

    res = pthread_create(&thread1, NULL, thread_function1, NULL);
    if (res != 0) {
        perror("Criação da Thread falhou");
        exit(EXIT_FAILURE);
    }
    printf("Criação da Thread1 \n");

    res = pthread_create(&thread2, NULL, thread_function2, NULL);
    if (res != 0) {
        perror("Criação da Thread falhou");
        exit(EXIT_FAILURE);
    }
    printf("Criação da Thread2 \n");

    res = pthread_join(thread1, &thread_result);
    if (res != 0) {
        perror("Junção da Thread1 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() --> Thread1 foi juntada com sucesso\n");

    res = pthread_join(thread2, &thread_result);
    if (res != 0) {
        perror("Junção da Thread2 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() --> Thread2 foi juntada com sucesso\n");

    pthread_mutex_destroy(&work_mutex); // destruição do mutex
    exit(EXIT_SUCCESS);
}

```

## Exercício: programa thread07.cpp

```

void *thread_function1(void *arg) {
    while(1)
    {
        printf("Thread1 -- Fora da região crítica \n");
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        printf("Thread1 -- Dentro da região crítica \n");
        sleep(1);
        pthread_mutex_unlock(&work_mutex);
    }
    pthread_exit(0);
}

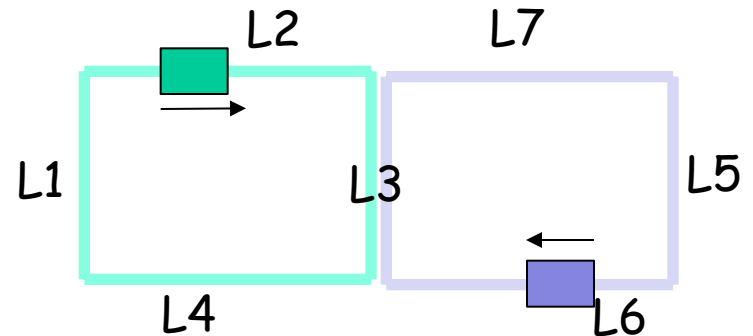
void *thread_function2(void *arg) {
    while(1)
    {
        printf("Thread2 -- Fora da região crítica \n");
        sleep(1);
        pthread_mutex_lock(&work_mutex);
        printf("Thread2 -- Dentro da região crítica \n");
        sleep(1);
        pthread_mutex_unlock(&work_mutex);
    }
    pthread_exit(0);
}

```

# Mutex

## ❑ Exercício: uso de mutex na Pthread

- Compile e execute o programa thread07.cpp
  - `g++ -o thread07 thread07.cpp -lpthread`
  - `./thread07`
  - Observe o código e analise o resultado quando se modifique os valores de tempo de sleeps.

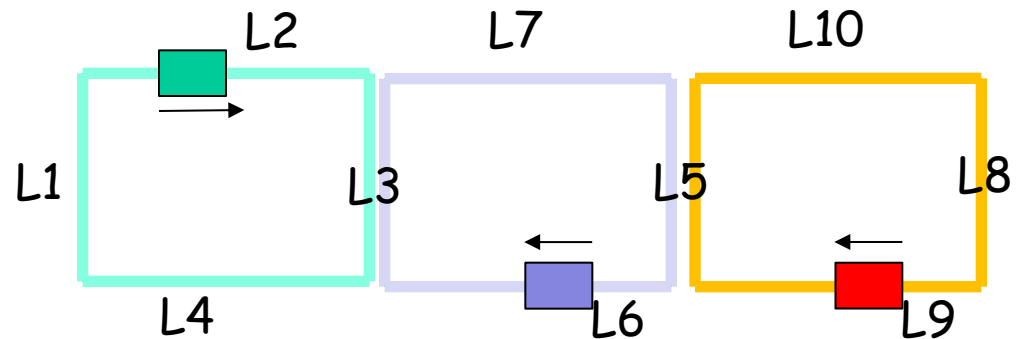


```
...  
while(1) {  
    L1();  
    L2();  
    lock(flag);  
    L3();  
    unlock(flag);  
    L4();  
}  
...
```

```
...  
while(1) {  
    L5();  
    L6();  
    lock(flag);  
    L3();  
    unlock(flag);  
    L7();  
}  
...
```

# Mutex

- ❑ Exercício: uso de mutex na Pthread
  - Implemente um programa para resolver a seguinte problema de programação concorrente.



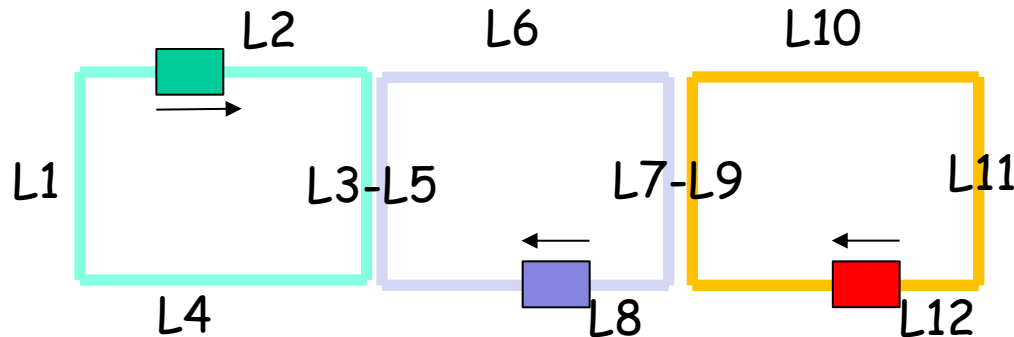
```
...  
while(1) {  
  
    ?????  
  
}  
...
```

```
...  
while(1) {  
  
    ????  
  
}  
...
```

```
...  
while(1) {  
  
    ???????  
  
}  
...
```

# Uso de mutex

□ Exercício: 3 Trens circulando com controle.



```
...  
while(1) {  
    L1();  
    L2();  
    lock(flag1);  
    L3();  
    unlock(flag);  
    L4();  
}  
...
```

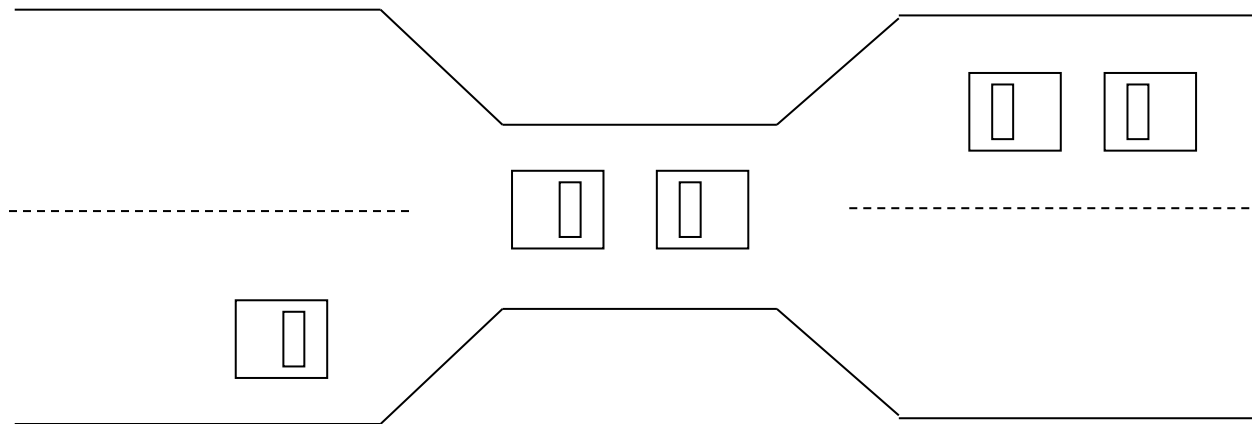
```
...  
while(1) {  
    L8();  
    ???  
}  
...
```

```
...  
while(1) {  
    L11();  
    L12();  
    lock(flag2);  
    L9();  
    unlock(flag2);  
    L(10);  
}  
...
```



# DeadLock

- ❑ Travamento perpétuo:
  - Problema inerente em sistemas concorrentes.
- ❑ Situação na qual um, ou mais processos, fica eternamente impedido de prosseguir sua execução devido ao fato de cada um estar aguardando acesso a recursos já alocados por outro processo.



# Condições para Haver Deadlock

## 1. Exclusão mútua

- Todo recurso ou está disponível ou está atribuído a um processo.

## 2. Segura/espera

- Os processos que detém um recurso podem solicitar novos recursos.

## 3. Recurso não-preemptível

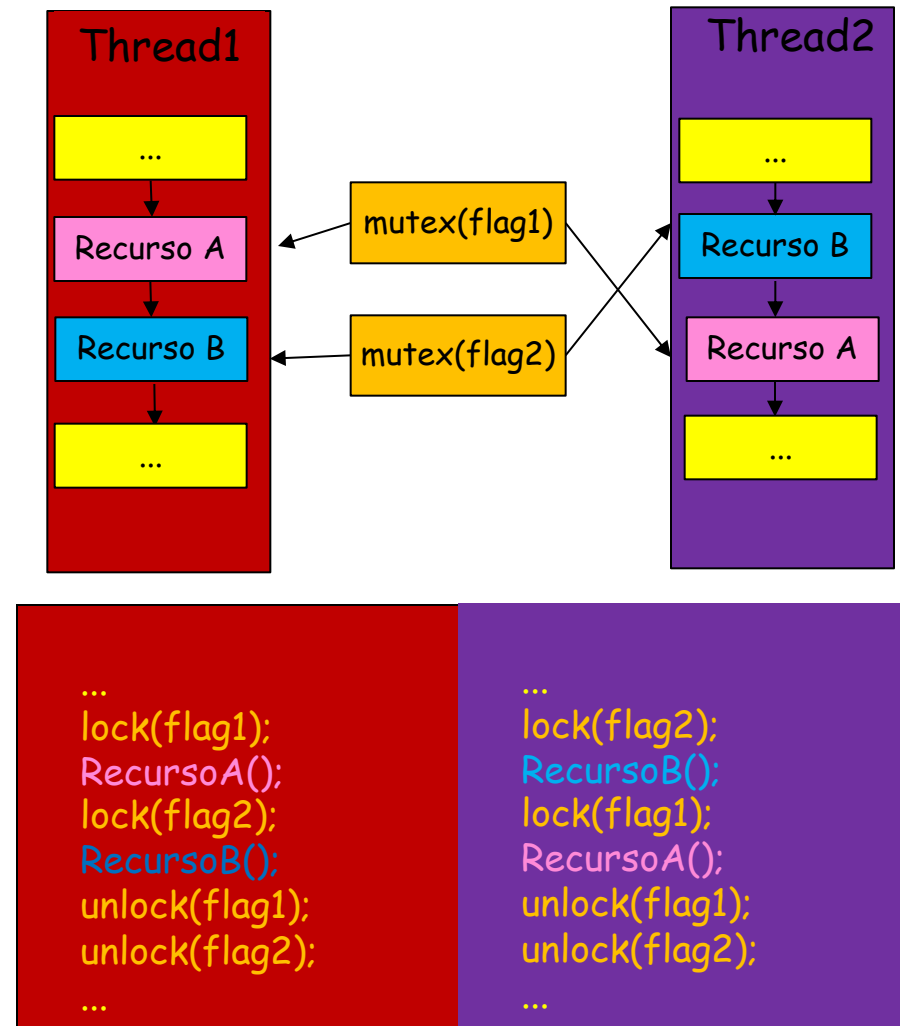
- Um recurso concedido não pode ser retirado de um processo por outro.

## 4. Espera Circular

- Existência de um ciclo de 2 ou mais processos, cada uma esperando por um recurso já adquirido (em uso) pelo próximo processo no ciclo.

# Deadlock

- ❑ Exercício: uso de semáforo Posix
  - Compile e execute o programa `thread_deadlock.cpp`
    - `g++ -o thread_deadlock thread_deadlock.cpp -lpthread`
    - `./thread_deadlock`
    - Observe o comportamento do programa.



## Exercício: programa thread\_deadlock.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

void *thread_function1(void *arg);
void *thread_function2(void *arg);

pthread_mutex_t mutex1, mutex2; /* declaração de um mutex */

int main() {
    int res;
    pthread_t thread1, thread2; // declaração de 02 threads
    void *thread_result;

    res = pthread_mutex_init(&mutex1, NULL); // criação do mutex1
    if (res != 0) {
        perror("Iniciação do Mutex1 falhou");
        exit(EXIT_FAILURE);
    }

    res = pthread_mutex_init(&mutex2, NULL); // criação do mutex2
    if (res != 0) {
        perror("Iniciação do Mutex2 falhou");
        exit(EXIT_FAILURE);
    }

    res = pthread_create(&thread1, NULL, thread_function1, NULL);
    if (res != 0) {
        perror("Criação da Thread falhou");
        exit(EXIT_FAILURE);
    }
    printf("Main() -- Criação da Thread1 \n");

    res = pthread_create(&thread2, NULL, thread_function2, NULL);
    if (res != 0) {
        perror("Criação da Thread falhou");
        exit(EXIT_FAILURE);
    }
    printf("Main() -- Criação da Thread2 \n");

    res = pthread_join(thread1, &thread_result);
    if (res != 0) {
        perror("Junção da Thread1 falhou");
        exit(EXIT_FAILURE);
    }
    printf("Main() -- Thread1 foi juntada com sucesso\n");

    res = pthread_join(thread2, &thread_result);
    if (res != 0) {
        perror("Junção da Thread2 falhou");
        exit(EXIT_FAILURE);
    }
    printf("Main() -- Thread2 foi juntada com sucesso\n");

    pthread_mutex_destroy(&mutex1); // destruição do mutex1
    pthread_mutex_destroy(&mutex2); // destruição do mutex2

    exit(EXIT_SUCCESS);
}
```

```
void *thread_function1(void *arg) {
    sleep(1);
    while(1)
    {
        printf("Thread1 -- Fora das Regiões Críticas A e B \n");
        sleep(1);
        printf("Thread1 -- Vai entrar na Região Crítica A\n");
        pthread_mutex_lock(&mutex1);
        printf("Thread1 -- Dentro da Região Crítica A \n");
        sleep(1);
        printf("Thread1 -- Vai entrar na Região Crítica B \n");
        pthread_mutex_lock(&mutex2);
        printf("Thread1 -- Dentro das Regiões Críticas A e B \n");
        sleep(1);
        pthread_mutex_unlock(&mutex1);
        printf("Thread1 -- Dentro da Região Crítica B e Fora da Região Crítica A\n");
        pthread_mutex_unlock(&mutex2);
    }
    pthread_exit(0);
}

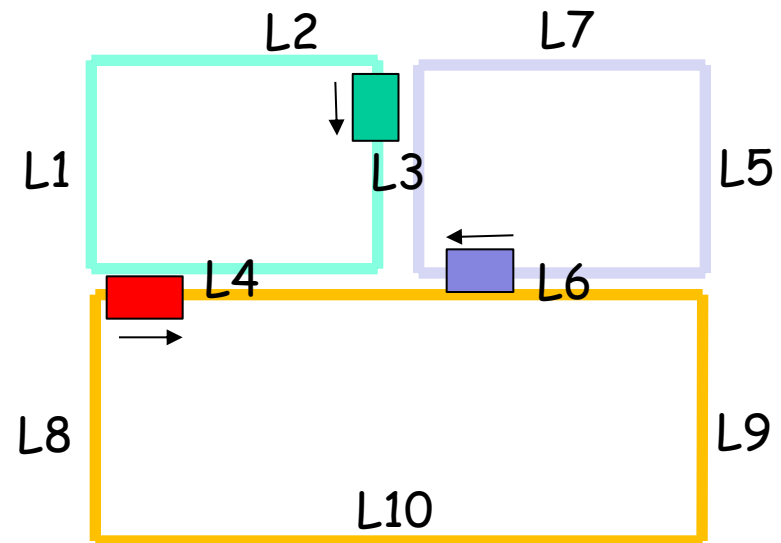
void *thread_function2(void *arg) {
    sleep(2);
    while(1)
    {
        printf("Thread2 -- Fora das Regiões Críticas A e B \n");
        sleep(1);
        printf("Thread2 -- Vai entrar na Região Crítica B\n");
        pthread_mutex_lock(&mutex2);
        printf("Thread2 -- Dentro da Região Crítica B\n");
        sleep(1);
        printf("Thread2 -- Vai entrar na Região Crítica A\n");
        pthread_mutex_lock(&mutex1);
        printf("Thread2 -- Dentro das Regiões Críticas A e B \n");
        sleep(1);
        pthread_mutex_unlock(&mutex2);
        printf("Thread2 -- Dentro da Região Crítica A e Fora da Região Crítica B\n");
        pthread_mutex_unlock(&mutex1);
    }
    pthread_exit(0);
}
```

# Evitando Deadlock

## ❑ Exercício:

- Implemente um programa para resolver a seguinte problema de programação concorrente.

- Os trens não podem colidir e nem haver deadlock.
- São necessários quantos mutexes (semáforos)?



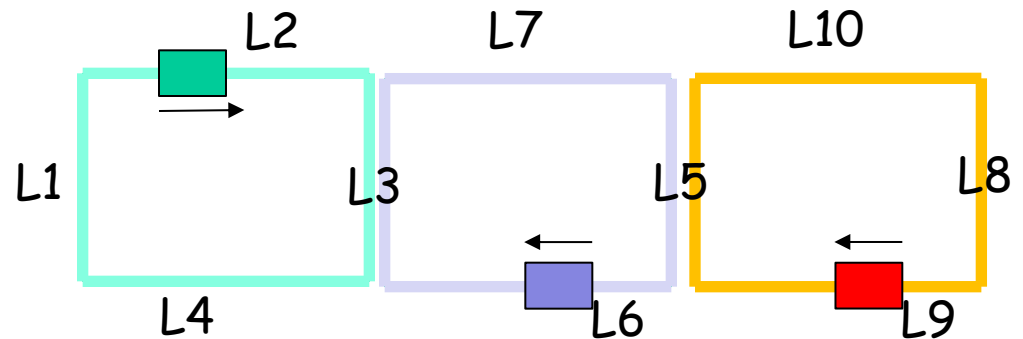
```
...  
while(1) {  
    L1();  
    L2();  
    L3();  
    L4();  
}  
...
```

```
...  
while(1) {  
    L5();  
    L6();  
    L3();  
    L7();  
}  
...
```

```
...  
while(1) {  
    L8();  
    L4();  
    L6();  
    L9();  
    L10();  
}  
...
```

# Mutex

- ❑ Exercício: uso de mutex na Pthread
  - Implemente um programa para resolver a seguinte problema de programação concorrente.

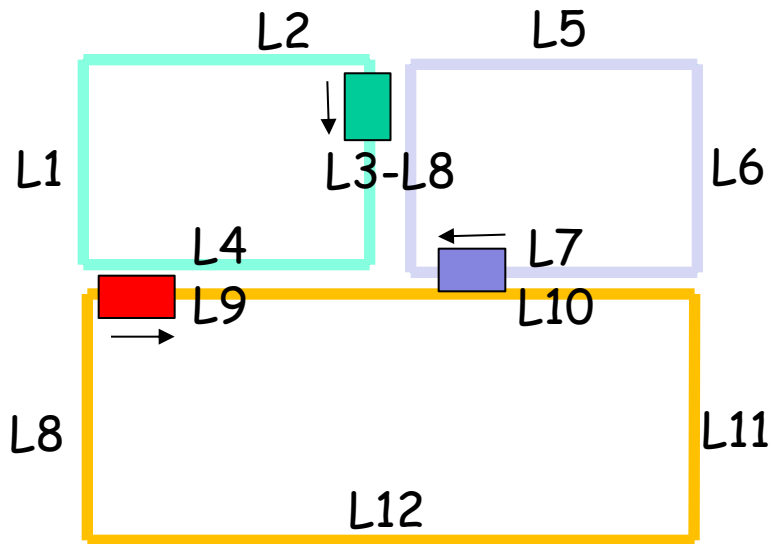


```
...  
while(1) {  
  
    ?????  
  
}  
...
```

```
...  
while(1) {  
  
    ????  
  
}  
...
```

```
...  
while(1) {  
  
    ???????  
  
}  
...
```

# Deadlock Problema dos 3 trens



```
...  
while(1) {  
    L1();  
    L2();  
    Lock(flag1);  
    L3();  
    Lock(flag2);  
    Unlock(flag1);  
    L4();  
    Unlock(flag2);  
}  
...
```

```
...  
while(1) {  
    L5();  
    L6();  
    Lock(flag3);  
    L7();  
    Lock(flag1);  
    Unlock(flag3);  
    L8();  
    Unlock(flag1);  
}  
...
```

```
...  
while(1) {  
    L12();  
    L8();  
    Lock(flag2);  
    L9();  
    Lock(flag3);  
    Unlock(flag2);  
    L10();  
    L11();  
    Unlock(flag3);  
}  
...
```

# Semáforos

- ❑ Mecanismo proposto por Dijkstra (1965)
- ❑ Semáforo é um tipo abstrato de dados:
  - Um valor inteiro não negativo
  - Uma fila FIFO de processos
- ❑ Há apenas duas operações atômicas:
  - P(Proberem, Down, **Wait**, Testar)
  - V(Verhogen, Up, **Post**, Incrementar)



# Semáforos

- ❑ Operações Atômicas  $V(s)$  e  $P(s)$ , sobre um semáforo  $s$ .
  - Semáforo binário:  $s$  só assume 0 ou 1.
  - Semáforo contador:  $s \geq 0$ .

## Primitivas P e V

```
P(s): s.valor = s.valor - 1
      Se s.valor < 0 {
        Bloqueia processo (sleep);
        Insere processo em S.fila;
      }
```

```
V(s): s.valor = s.valor + 1
      Se S.valor <= 0 {
        Retira processo de S.fila;
        Acorda processo (wakeup);
      }
```

# Usando Semáforo para Exclusão Mútua

A iniciação do semáforo **s** é efetuada em um dos processos

Processo 1

```
...  
s = 1;  
...  
P(s);  
Regiao_critica();  
V(s);  
Regiao_nao_critica();  
...
```

Processo 2

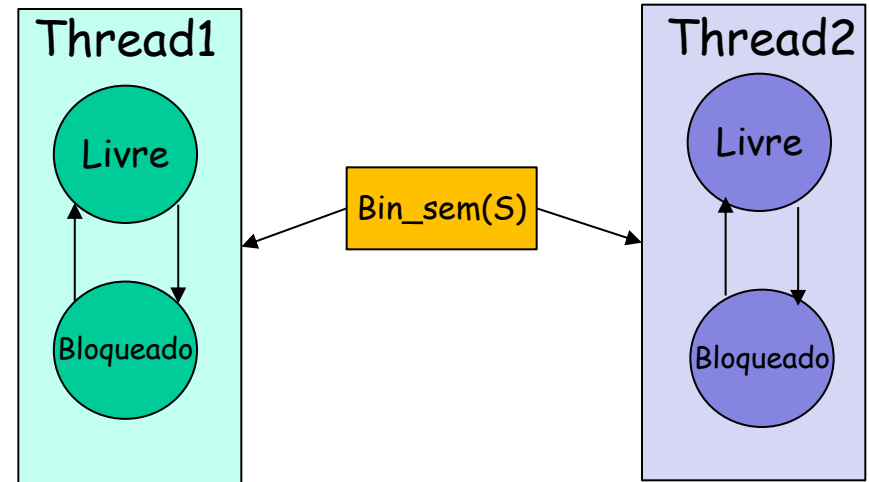
```
...  
...  
P(s);  
Regiao_critica();  
V(s);  
Regiao_nao_critica();  
...
```

# Semáforo Binário

## ❑ Exercício: uso de semáforo Posix

- Compile e execute o programa `thread_semaforo_binario.cpp`

- `g++ -o thread_semaforo_binario thread_semaforo_binario.cpp -lpthread`
- `./thread_semaforo_binario`
- Observe o código e analise o resultado quando se modifique os valores de tempo de sleeps.



```
S = 1;  
...  
wait(S);  
Regiao_critica();  
post(S);  
Regiao_nao_critica();  
...
```

```
...  
wait(S);  
Regiao_critica();  
post(s);  
Regiao_nao_critica();  
...
```

```

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>

void *thread_function1(void *arg);
void *thread_function2(void *arg);
sem_t *bin_sem;

int main() {
    int res = 0;
    int value;
    pthread_t thread1, thread2; // declaração de 02 threads
    void *thread_result;

    res = sem_init(&bin_sem, 0, 1);
    if (res != 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }

    res = pthread_create(&thread1, NULL, thread_function1, NULL);
    if (res != 0) {
        perror("Criação da Thread1 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() -- Criação da Thread1 \n");

    res = pthread_create(&thread2, NULL, thread_function2, NULL);
    if (res != 0) {
        perror("Criação da Thread2 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() -- Criação da Thread2 \n");

    res = pthread_join(thread1, &thread_result);
    if (res != 0) {
        perror("Junção da Thread1 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() -- Thread1 foi juntada com sucesso\n");

    res = pthread_join(thread2, &thread_result);
    if (res != 0) {
        perror("Junção da Thread2 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() -- Thread2 foi juntada com sucesso\n");
    printf("MAIN() -- A THREAD MAIN() vai terminar\n");

    sem_destroy(&bin_sem);
    exit(EXIT_SUCCESS);
}

```

## Exercício: programa thread\_semaforo\_binario.cpp

```

void *thread_function1(void *arg) {
    sleep(1);
    while(1)
    {
        printf("Thread1 -- Fora da região crítica \n");
        sleep(1);
        sem_wait(&bin_sem);
        printf("Thread1 -- Dentro da região crítica \n");
        sleep(1);
        sem_post(&bin_sem);
    }
    pthread_exit(0);
}

void *thread_function2(void *arg) {
    sleep(1);
    while(1)
    {
        printf("Thread2 -- Fora da região crítica \n");
        sleep(1);
        sem_wait(&bin_sem);
        printf("Thread2 -- Dentro da região crítica \n");
        sleep(5);
        sem_post(&bin_sem);
    }
    pthread_exit(0);
}

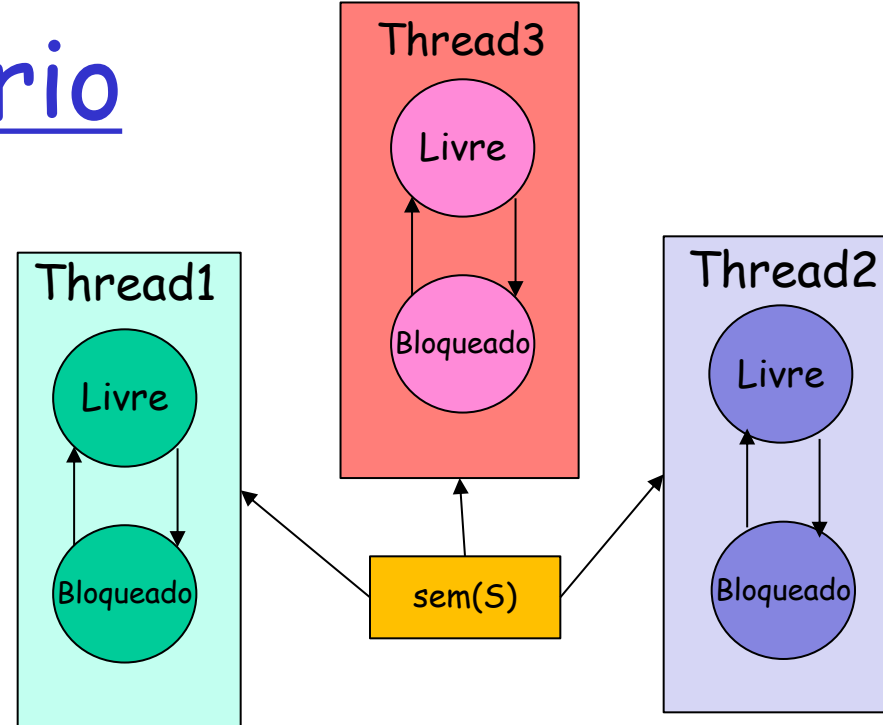
```

# Semáforo Não-Binário

## ❑ Exercício: uso de semáforo Posix

- Compile e execute o programa `thread_semaforo_naobinario.cpp`

- `g++ -o thread_semaforo_naobinario thread_semaforo_naobinario.cpp -lpthread`
- `./thread_semaforo_naobinario`
- Observe o código e analise o resultado quando se modifique os valores de tempo de sleeps e o valor inicial do semáforo.



```
...  
S = ?;  
...  
wait(S);  
Regiao_critica();  
post(S);  
Regiao_nao_critica();  
...
```

```
...  
wait(S);  
Regiao_critica();  
post(s);  
Regiao_nao_critica();  
...
```

```
...  
wait(S);  
Regiao_critica();  
post(s);  
Regiao_nao_critica();  
...
```

## Exercício: programa thread\_semaforo\_naobinario.cpp

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>

void *thread_function1(void *arg);
void *thread_function2(void *arg);
void *thread_function3(void *arg);
sem_t *not_bin_sem;

int main() {
    int res;
    pthread_t thread1, thread2, thread3; // declaração de 03 threads
    void *thread_result;

    res = sem_init(&not_bin_sem, 0, 2);
    if (res < 0) {
        perror("Semaphore initialization failed");
        exit(EXIT_FAILURE);
    }

    res = pthread_create(&thread1, NULL, thread_function1, NULL);
    if (res != 0) {
        perror("Criação da Thread1 falhou");
        exit(EXIT_FAILURE);
    }
    printf("Criação da Thread1 \n");

    res = pthread_create(&thread2, NULL, thread_function2, NULL);
    if (res != 0) {
        perror("Criação da Thread2 falhou");
        exit(EXIT_FAILURE);
    }
    printf("Criação da Thread2 \n");

    res = pthread_create(&thread3, NULL, thread_function3, NULL);
    if (res != 0) {
        perror("Criação da Thread3 falhou");
        exit(EXIT_FAILURE);
    }
    printf("Criação da Thread3 \n");

    res = pthread_join(thread1, &thread_result);
    if (res != 0) {
        perror("Junção da Thread1 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() --> Thread1 foi juntada com sucesso\n");

    res = pthread_join(thread2, &thread_result);
    if (res != 0) {
        perror("Junção da Thread2 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() --> Thread2 foi juntada com sucesso\n");

    res = pthread_join(thread3, &thread_result);
    if (res != 0) {
        perror("Junção da Thread3 falhou");
        exit(EXIT_FAILURE);
    }
    printf("MAIN() --> Thread3 foi juntada com sucesso\n");

    printf("MAIN() --> A THREAD main vai terminar\n");

    sem_destroy(&not_bin_sem);
    exit(EXIT_SUCCESS);
}
```

```
void *thread_function1(void *arg) {
    sleep(1);
    while(1)
    {
        printf("Thread1 -- Fora da região crítica \n");
        sleep(1);
        sem_wait(&not_bin_sem);
        printf("Thread1 -- Dentro da região crítica \n");
        sleep(1);
        sem_post(&not_bin_sem);
    }
    pthread_exit(0);
}

void *thread_function2(void *arg) {
    sleep(1);
    while(1)
    {
        printf("Thread2 -- Fora da região crítica \n");
        sleep(1);
        sem_wait(&not_bin_sem);
        printf("Thread2 -- Dentro da região crítica \n");
        sleep(1);
        sem_post(&not_bin_sem);
    }
    pthread_exit(0);
}

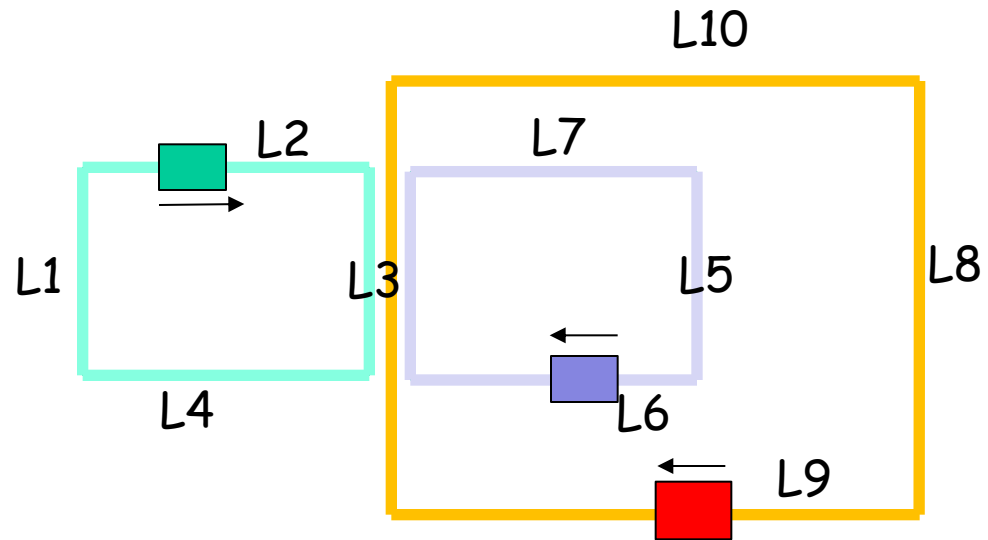
void *thread_function3(void *arg) {
    sleep(1);
    while(1)
    {
        printf("Thread3 -- Fora da região crítica \n");
        sleep(1);
        sem_wait(&not_bin_sem);
        printf("Thread3 -- Dentro da região crítica \n");
        sleep(10);
        sem_post(&not_bin_sem);
    }
    pthread_exit(0);
}
```

# Semáforo

## ❑ Exercício: uso de semáforo na Pthread

- Implemente um programa para resolver a seguinte problema de programação concorrente.

- É possível haver até 2 trens no trilho L3 ao mesmo tempo.



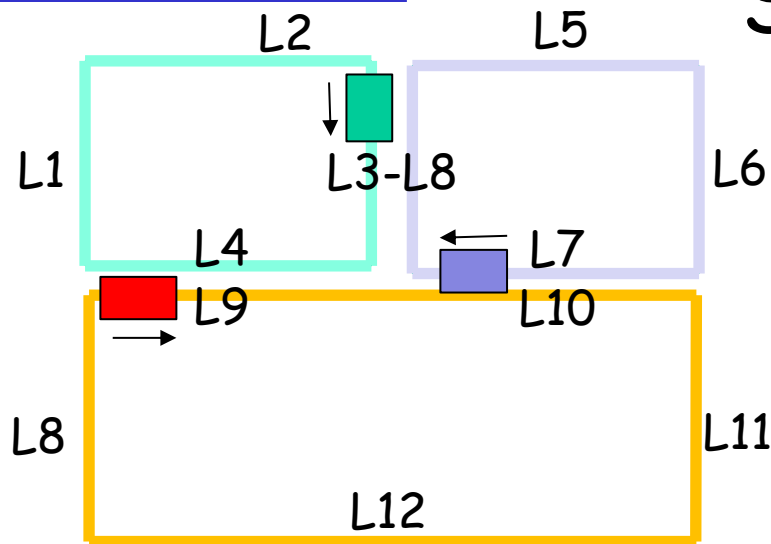
```
S1 = ?;  
while(1) {  
    L1();  
    L2();  
    wait(S1);  
    L3();  
    post(S1);  
    L4();  
}  
...
```

```
...  
while(1) {  
    L5();  
    L6();  
    wait(S1);  
    L3();  
    post(S1);  
    L7();  
}  
...
```

```
...  
while(1) {  
    L8();  
    L9();  
    wait(S1);  
    L3();  
    post(S1);  
    L10();  
}  
...
```

# Deadlock

## Problema dos 3 trens com Semáforo Não Binário



```
...  
while(1) {  
    L1();  
    L2();  
    wait(S1);  
    Lock(flag1);  
    L3();  
    Lock(flag2);  
    post(S1);  
    Unlock(flag1);  
    L4();  
    Unlock(flag2);  
}  
...
```

```
...  
while(1) {  
    L5();  
    L6();  
    wait(S1);  
    Lock(flag3);  
    L7();  
    Lock(flag1);  
    post(S1);  
    Unlock(flag3);  
    L8();  
    Unlock(flag1);  
}  
...
```

```
...  
S1 = 2;  
while(1) {  
    L12();  
    L8();  
    wait(S1);  
    Lock(flag2);  
    L9();  
    Lock(flag3);  
    post(S1);  
    Unlock(flag2);  
    L10();  
    L11();  
    Unlock(flag3);  
}  
...
```



# Semáforo X Mutex

## ❑ Mutex

- As operações `lock()` e `unlock()` devem ser executadas necessariamente pelo mesmo processo.

## ❑ Semáforos

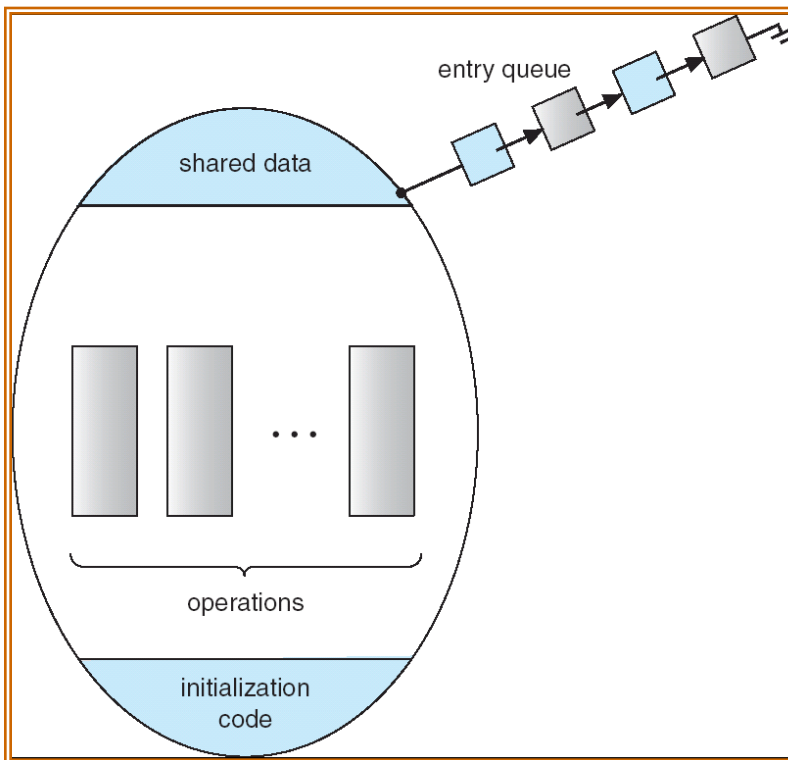
- As primitivas `V(s)` [`wait(s)`] e `P(s)` [`post(s)`] podem ser executadas por processos diferentes
- `S` pode assumir valor maior que 1.
  - Gerência de recursos
  - Mais geral que mutex

# Monitores

- ❑ Primitiva de alto nível para sincronização de processo.
- ❑ Bastante adequado para programação orientada a objetos.
- ❑ Somente um processo pode estar ativo dentro de um monitor de cada.

# Sintaxe de um Monitor

- Um monitor agrupar várias funções mutuamente excludentes.



```
monitor monitor name
{
    // shared variable declarations

    initialization code ( . . . ) {
        . . .
    }

    public P1 ( . . . ) {
        . . .
    }

    public P2 ( . . . ) {
        . . .
    }
    .
    .
    .
    public Pn ( . . . ) {
        . . .
    }
}
```

# Monitores X Semáforos

- ❑ Monitores permitem estruturar melhor os programas
- ❑ Pode-se implementar Monitores através de Semáforos e vice-versa.
- ❑ Java inicialmente só implementava monitores.
  - Atualmente também possui Semáforos