Thread (Sincronia)

- Problemas:
  - **Condição de Corrida:**
    - Uma área de memória compartilhada é modificada por mais de uma *thread* e no meio de uma operação de leitura ou escrita, a *thread* é bloqueada.
  - **Deadlock**
    - Falha dos mecanismos de sincronia de *threads*

Thread (Sincronia)

- Condição de Corrida - Exemplo 1 : threads_concorrencia_01.c

```c
// Função que incrementa o Contador
void* incrementa_contador (void *arg) {
    for (unsigned int i=0; i < 100; i++) {
        puts("Inicia   ++");
        usleep(random() % 100000);
        puts("Finaliza ++");
    }
    return NULL;
}
// Função que decrementa o Contador
void* decrementa_contador (void *arg) {
    for (unsigned int i=0; i < 100; i++) {
        puts("Inicia   --");
        usleep(random() % 100000);
        puts("Finaliza --");
    }
    return NULL;
}
int main (int argc, char** argv) {
    pthread_t t0;
    pthread_t t1;
    int res0, res1;
    res0 = pthread_create(&t0, NULL, incrementa_contador, NULL);
    res1 = pthread_create(&t1, NULL, decrementa_contador, NULL);
    res0 = pthread_join(t0, NULL);
    res0 = pthread_join(t1, NULL);
    puts("Terminou!");
    return 0;
}
```

Thread (Sincronia)

- Condição de Corrida - Exemplo 2 : threads_concorrencia_02.c

variável compartilhada (Global)

```
volatile int varCompartilhada=0;
```

main

```
pthread_t t0, t1;
int res0, res1;
res0 = pthread_create(&t0, NULL, incrementa_contador, NULL);
res1 = pthread_create(&t1, NULL, decrementa_contador, NULL);
res0 = pthread_join(t0, NULL);
res0 = pthread_join(t1, NULL);
printf("Valor final: %d\n", varCompartilhada);
```

Thread 1

```
void* incrementa_contador (void *arg) {
    for (unsigned int i=0; i < 10000; i++) {
        varCompartilhada++;
    }
    return NULL;
}
```

Thread 2

```
void* decrementa_contador (void *arg) {
    for (unsigned int i=0; i < 10000; i++) {
        varCompartilhada--;
    }
    return NULL;
}
```

Thread (Sincronia)

- Condição de Corrida - Exemplo 2 : threads_concorrencia_02.c

variável compartilhada (Global)

```
volatile
pthre
int r
res0
res1
res0
res0
print
                                                    JLL);
                                                    JLL);
```

Intel Disassembly

varCompartilhada++

```
movl   _varCompartilhada(%rip), %eax
addl   $1, %eax
movl   %eax, _varCompartilhada(%rip)
```

varCompartilhada--

```
movl   _varCompartilhada(%rip), %eax
subl   $1, %eax
movl   %eax, _varCompartilhada(%rip)
```

Thread 1 **Thread 2**

```
void* incrementa_contador (void *arg) {
    for (unsigned int i=0; i < 10000; i++) {
        varCompartilhada++;
    }
    return NULL;
}
```

```
void* decrementa_contador (void *arg) {
    for (unsigned int i=0; i < 10000; i++) {
        varCompartilhada--;
    }
    return NULL;
}
```

Thread (Sincronia)

- Condição de Corrida - Exemplo 2 : threads_concorrencia_02.c

Thread 0 - **`cont++`**	Thread 1 - **`count--`**
reg1=cont	
reg1 = reg1 **+** 1	Bloqueado
cont = reg1	
Bloqueado	reg1=cont
	reg1 = reg1 **-** 1
	cont = reg1

Thread (Sincronia)

- Condição de Corrida - Exemplo 2 : threads_concorrencia_02.c

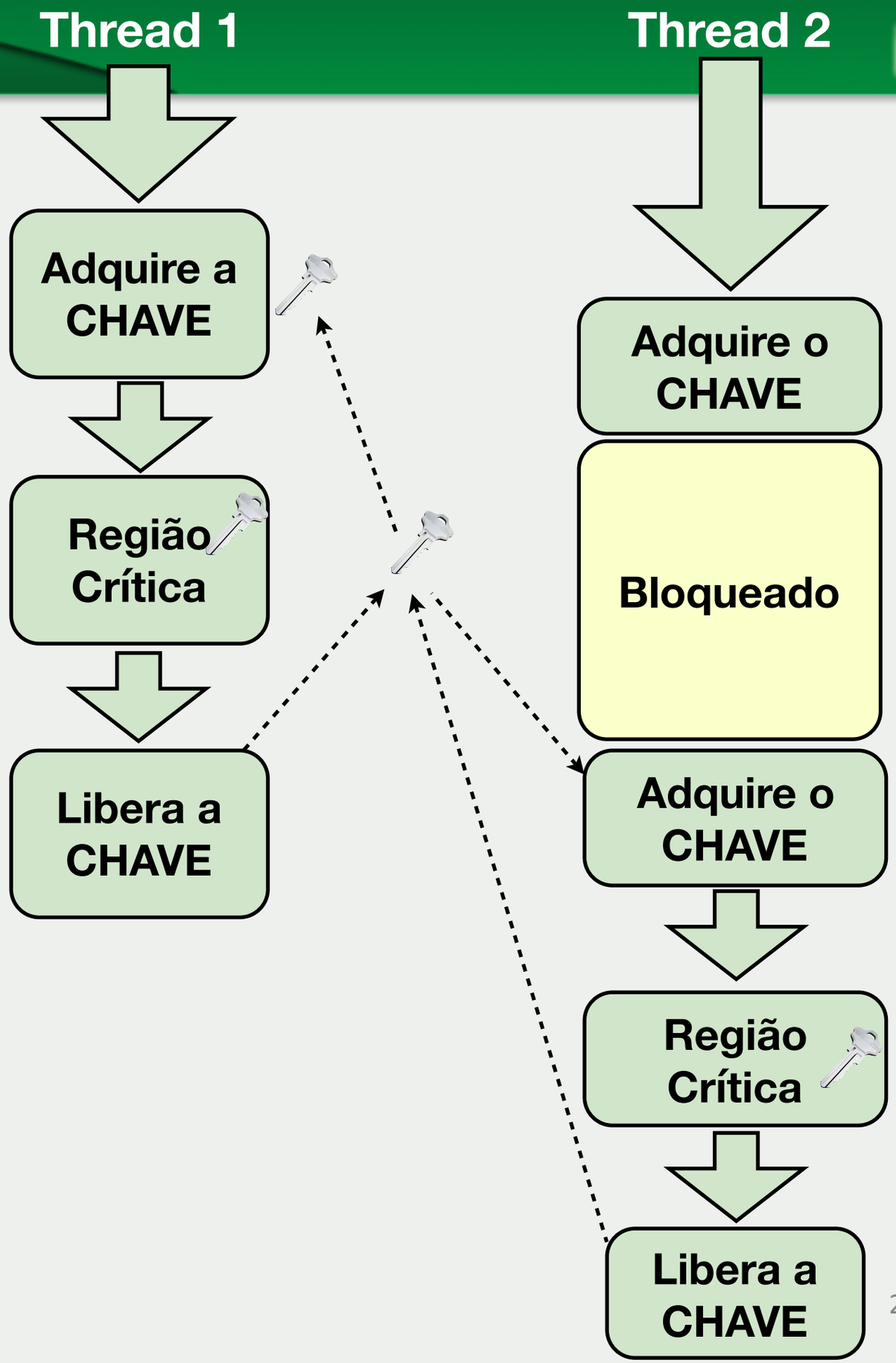
Thread 0 - **`cont++`**	Thread 1 - **`count--`**
reg1=cont	Bloqueado
reg1 = reg1 **+** 1	
Bloqueado	reg2=cont
	reg2 = reg2 **-** 1
	cont = reg2
cont = reg1	Bloqueado

Thread (Sincronia)

- Possíveis Soluções:
  - Mutex (Mutual Exclusive)
  - Semáforos

MUTEX

- Variável especial
- Valores possíveis: 0 ou 1
- Ações: “Adquirir” ou “Liberar” a chave
- Garantia para operações atômicas
- A Aquisição do MUTEX é blocante caso a chave esteja sendo usada.



MUTEX

Define variável global do MUTEX

```
pthread_mutex_t mutexLock;
```

Inicializa o MUTEX

```
res = pthread_mutex_init(&mutexLock, NULL);
```

Adquire a CHAVE (blocante)

```
pthread_mutex_lock(&mutexLock);
```

Libera a CHAVE

```
pthread_mutex_unlock(&mutexLock);
```

Destrói o MUTEX

```
pthread_mutex_destroy(&mutexLock);
```

Thread (Sincronia)

- MUTEX - Exemplo 3 : threads_concorrencia_03.c

variáveis compartilhadas (Global)

```
volatile int varCompartilhada=0;
static pthread_mutex_t mutexLock;
```

main

```
    pthread_t t0, t1;
    int res, res0, res1;
    res = pthread_mutex_init(&mutexLock, NULL);
    res0 = pthread_create(&t0, NULL, incrementa_contador, NULL);
    res1 = pthread_create(&t1, NULL, decrementa_contador, NULL);
    res0 = pthread_join(t0, NULL);
    res0 = pthread_join(t1, NULL);
    pthread_mutex_destroy(&mutexLock);
    printf("Valor final: %d\n", varCompartilhada);
```

Thread 1

```
void* incrementa_contador (void *arg) {
    for (unsigned int i=0; i < 10000; i++) {
        pthread_mutex_lock(&mutexLock);
        varCompartilhada++;
        pthread_mutex_unlock(&mutexLock);
    }
    return NULL;
}
```

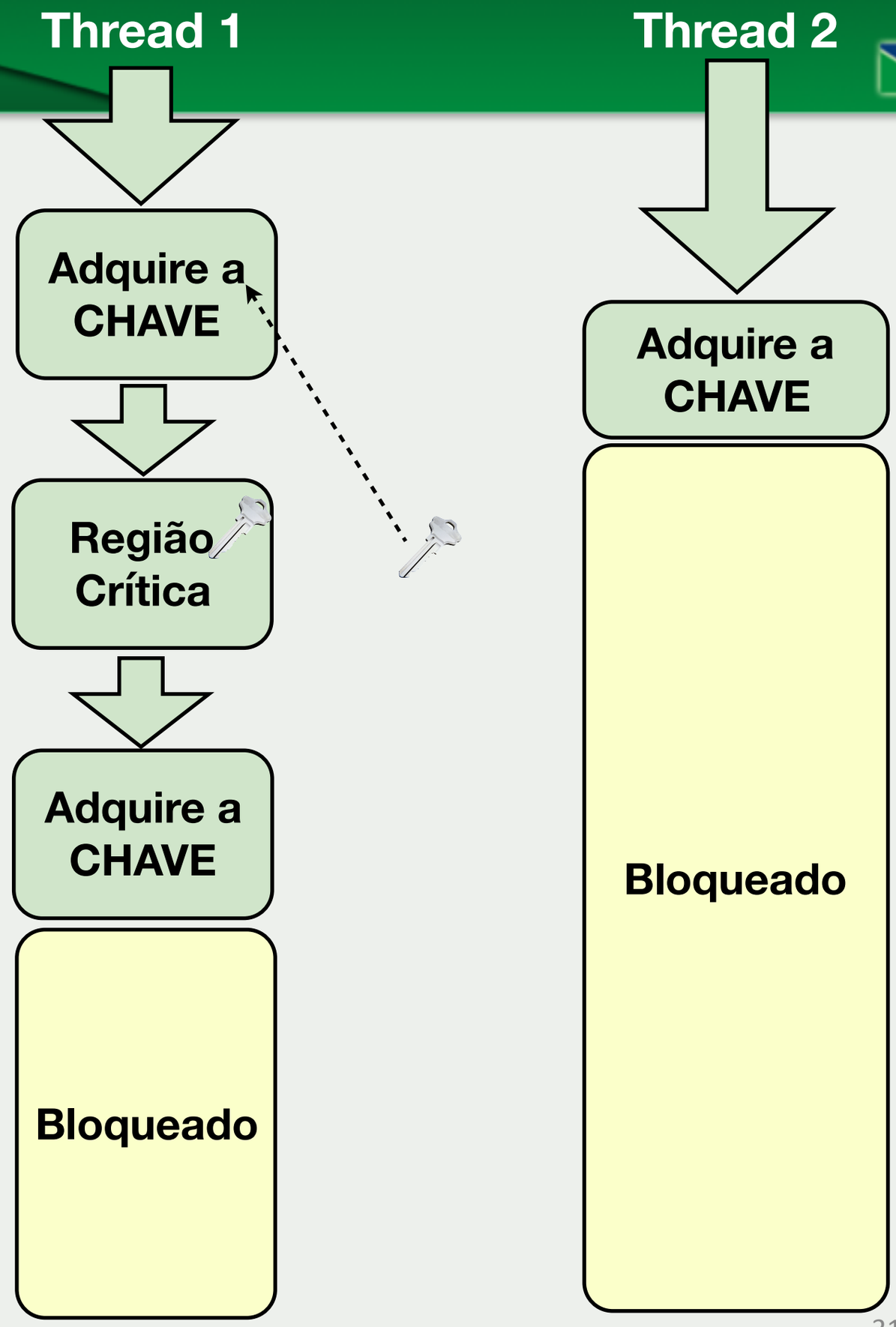
Thread 2

```
void* decrementa_contador (void *arg) {
    for (unsigned int i=0; i < 10000; i++) {
        pthread_mutex_lock(&mutexLock);
        varCompartilhada--;
        pthread_mutex_unlock(&mutexLock);
    }
    return NULL;
}
```

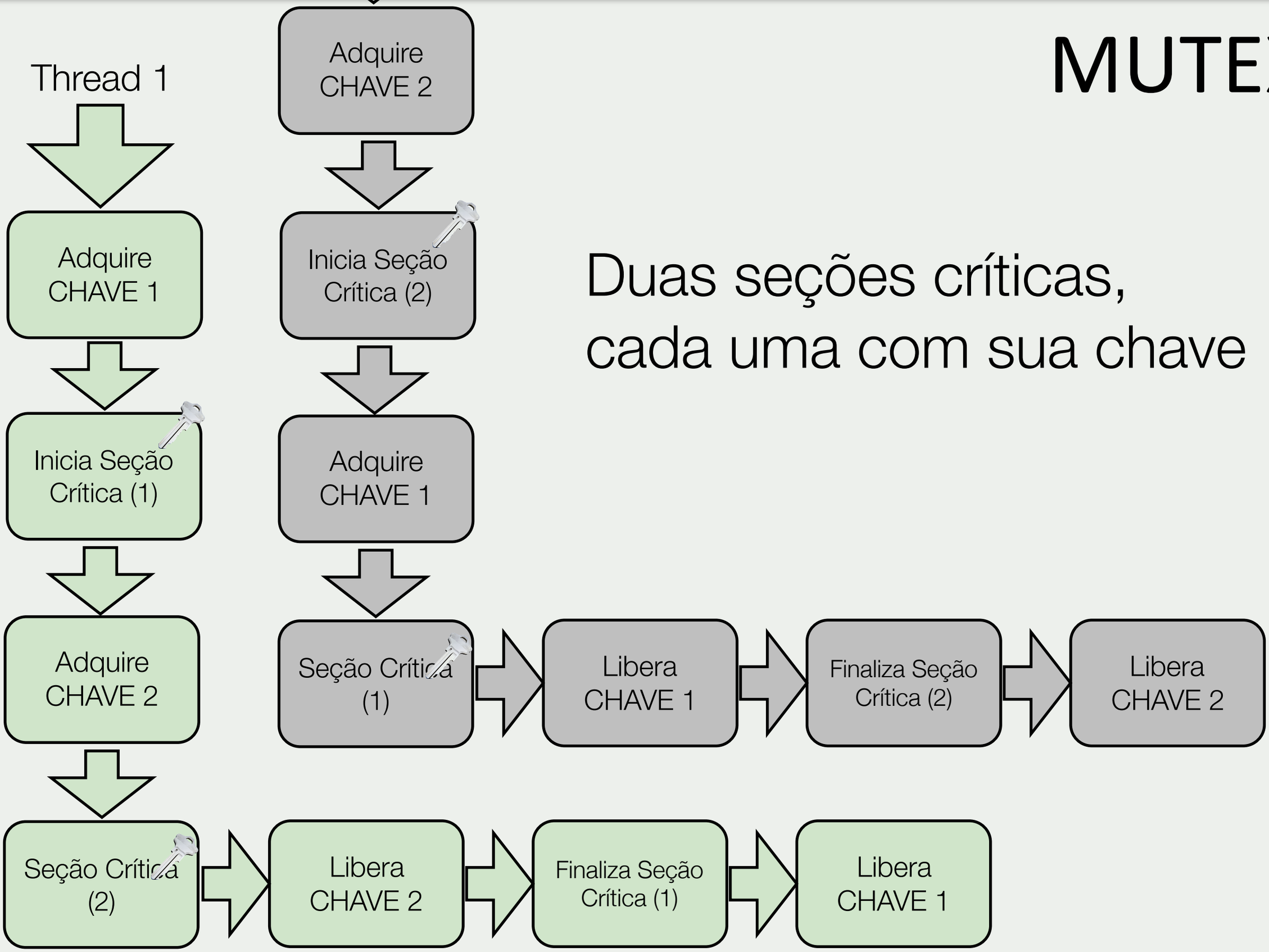
MUTEX

- Perigo!!!
- Caso a CHAVE não seja liberada.

DEADLOCK

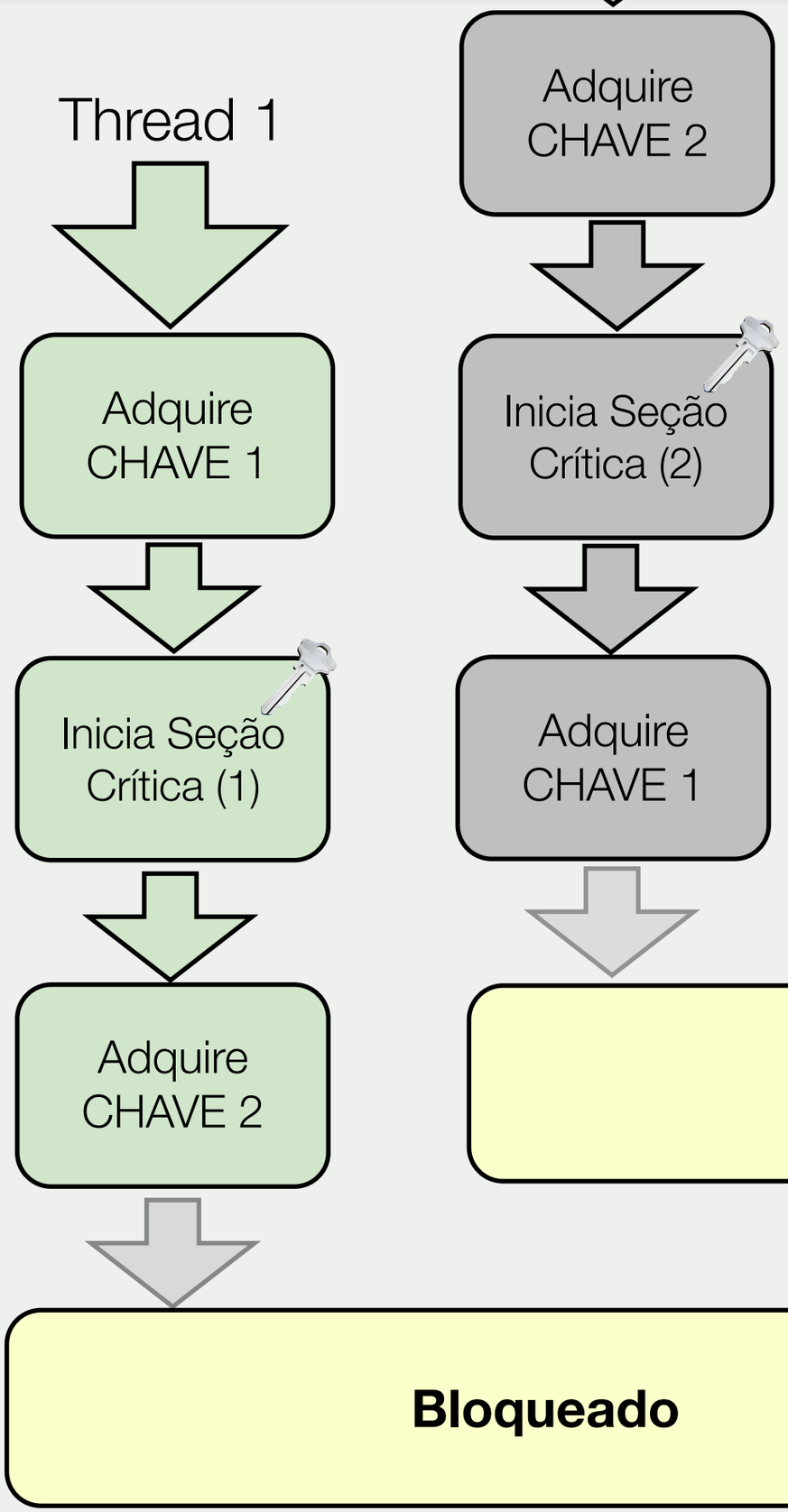


MUTEX



Duas seções críticas,
cada uma com sua chave

MUTEX



Agora mudando a ordem de execução…

DEADLOCK

MUTEX - Soluções para DEADLOCK

- É importante escrever o código de modo a tratar os DEADLOCKS mesmo achando que não irão ocorrer!
- Soluções:
  - **Watch Dog Timer**
    - Cada thread deve reportar que está “viva" periodicamente numa variável global.
    - Um *timer* verifica e zera esta variável global periodicamente. Caso a thread não tenha respondido, é terminada.
  - **Timeout para Threads e MUTEXES**
    - Todos os MUTEXes blocantes são liberados após um tempo determinado.