Sincronização de Sistemas Concorrentes
Parte I

Programação Avançada
Patrício Domingues

Programação concorrente

- ✓ "Concorrência" entre fluxos de execução
  - Dois ou mais fluxos de execução (e.g, *threads* ou processos), podem competir por um mesmo recurso
- ✓ Exemplo
  - Acesso a uma variável partilhada
    - Lembrete: **variável** é na realidade um valor guardado numa zona de memória
  - Cenário para concorrência
    - N *threads* acedem à variável
    - Pelo menos uma *thread* escreve (modifica) na variável
      - Se não existir sincronização no acesso à variável partilhada então existirá uma corrida por recurso (*race condition*)

Variável global e duas threads... (1)

- Considere a variável G_Total

```
int G_Total = 0;
```

- E a função Adiciona

```
int Adiciona(int add)
{
    G_Total += add;
}
```

- Considere as threads T1 e T2 que executam concorrentemente

```
T1:
   Adiciona(5);
T2:
   Adiciona(10);
```

Qual o resultado final?

Resposta: slide seguinte >>

Exemplo

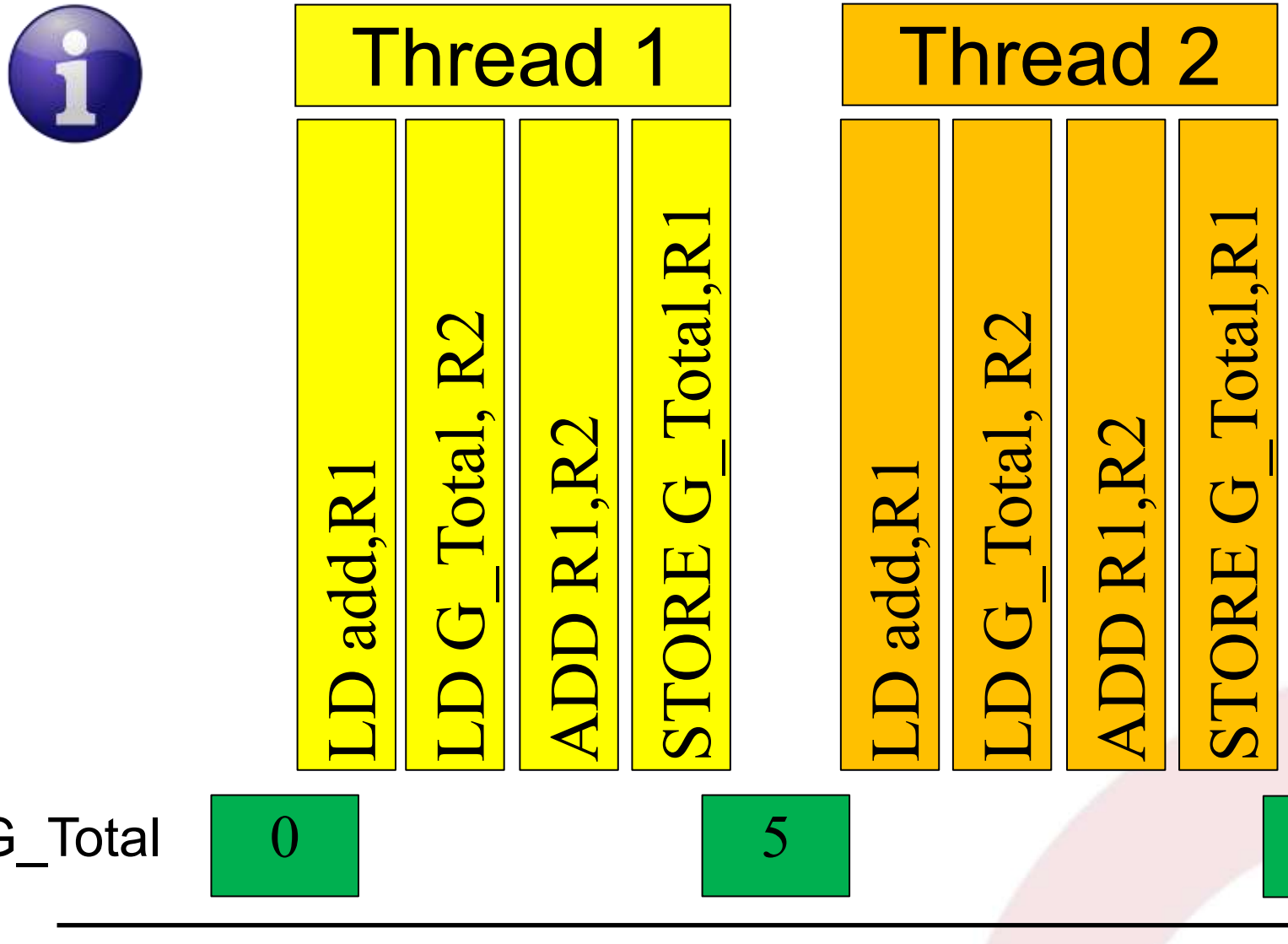
Variável global e duas threads... (2)

- ✓ Depende da ordem de execução das threads T1 e T2
- ✓ A operação de soma (G_Total +=add;) poderá ser mapeada para várias instruções de assembler

```
LD add, R1          /* Carrega registo R1 com "add" */
LD G_Total, R2      /* Carrega registo R2 com "G_Total" */
ADD R1,R2           /* Efetua a soma R1 = R1 + R2 */
STORE G_Total,R1    /* Guarda R1 (soma) para G_Total */
```

- ✓ Uma instrução "C" é mapeada para quatro instruções assembler
  - A instrução C não é atómica...

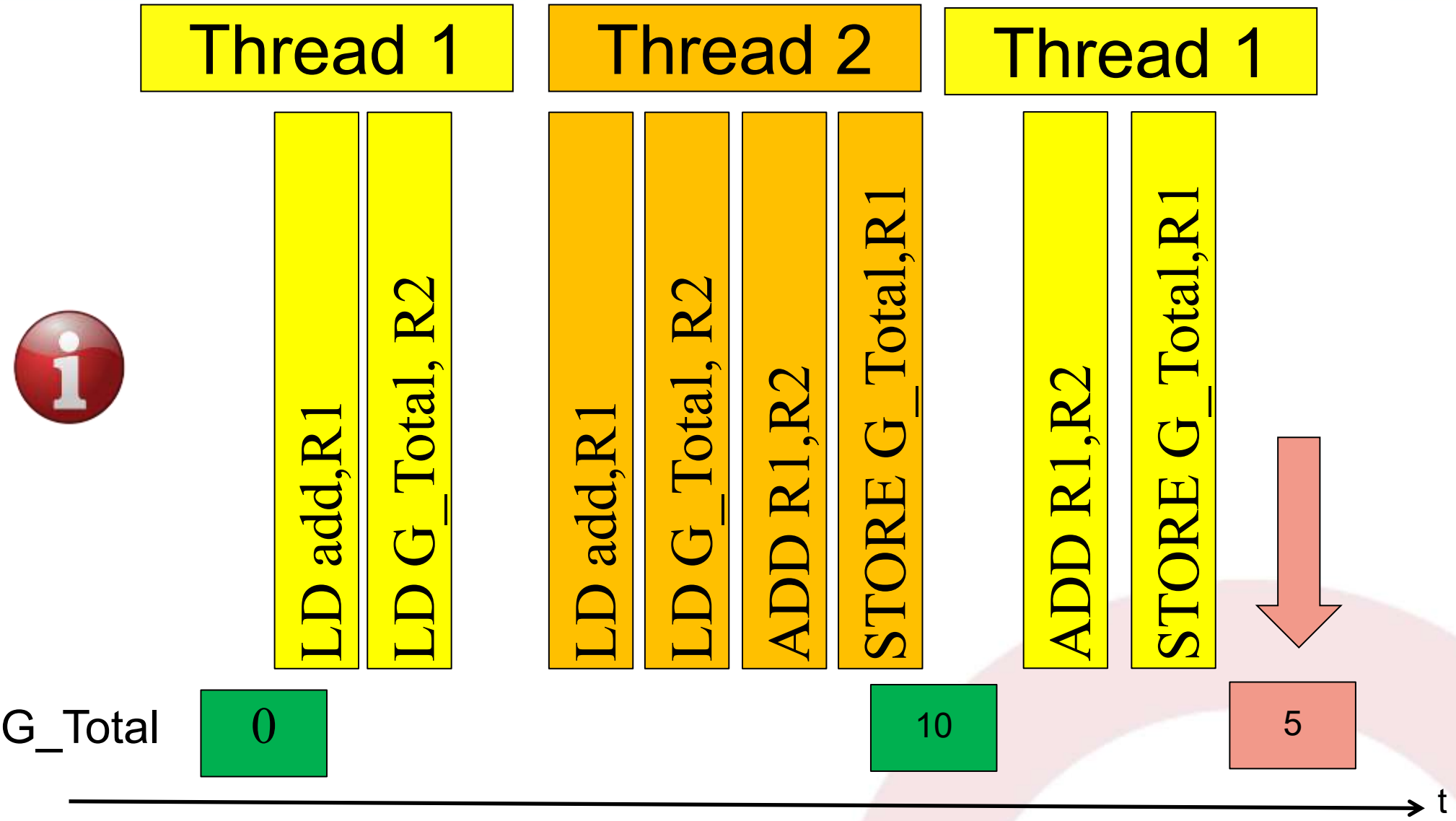
Variável global e duas threads... (3)



- Caso 1: executa a thread 1 e depois a thread 2 (ou vice versa)

Caso 2 >>

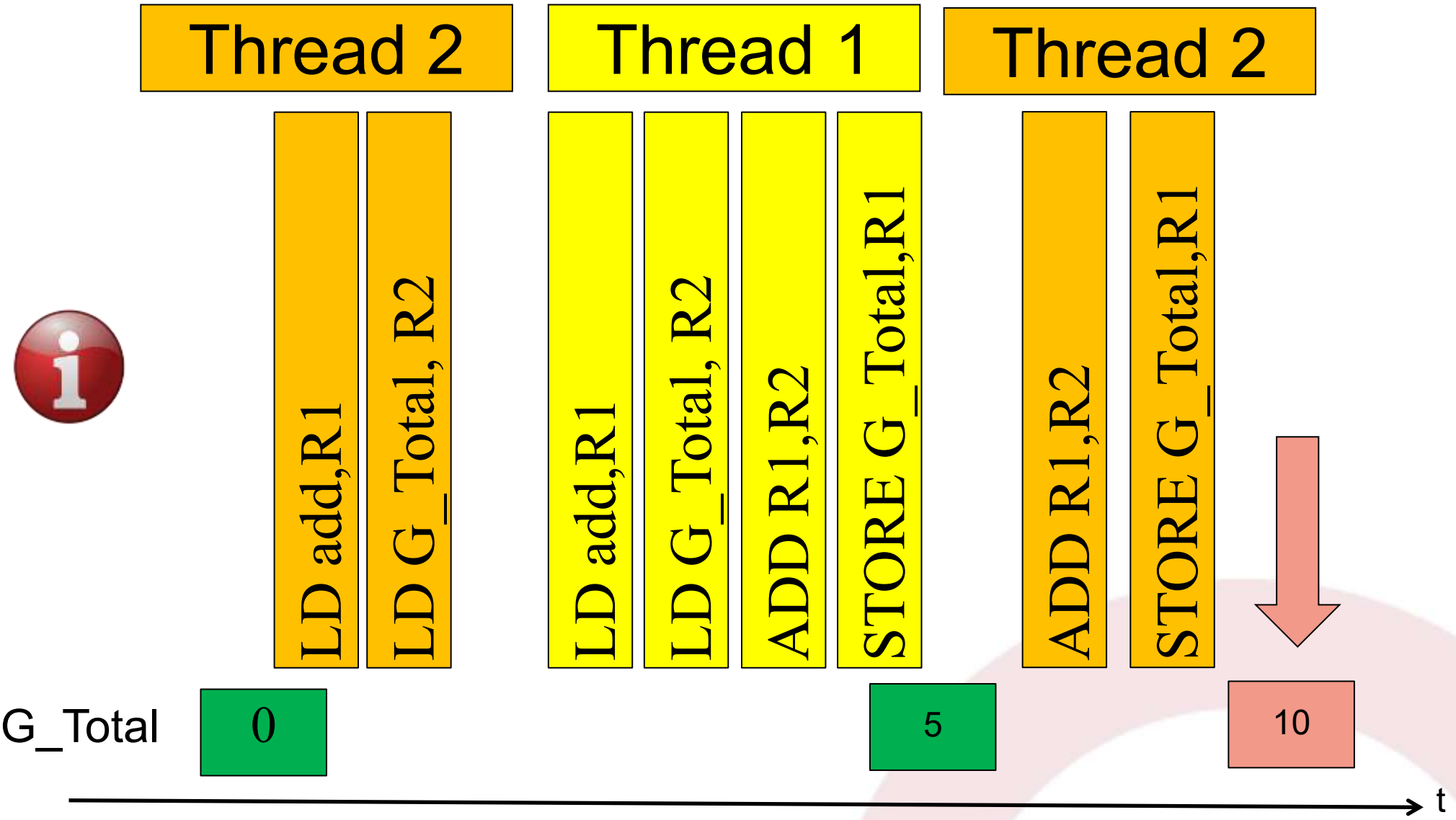
Variável global e duas threads... (4)



- Caso 2: executa parcialmente a *thread 1*, depois a *thread 2* (totalmente) e novamente o que resta da *thread 1*

Caso 3 >>

Variável global e duas threads... (4)



- Caso 2: executa parcialmente a *thread 2*, depois a *thread 1* (totalmente) e novamente o que resta da *thread 2*

Análise >>

Variável global e duas threads... (5)

- ✓ Análise
  - Quando T1 executa primeiro e depois T2, o resultado final é 15
  - Quando T2 executa primeiro e depois T1, o resultado final é 15
    - Comportamento esperado
- ✓ Mas…
  - Quando T1 inicia execução, é interrompido, T2 executa e finalmente, T1 termina a sua execução
    - Resultado final é 5
  - Quando T2 inicia execução, é interrompido, T1 executa e finalmente, T2 termina a sua execução
    - Resultado final é 10
- ✓ Resumindo
  - ✓ O resultado final depende da ordem de execução das duas threads
    - O código não está preparado para execução concorrente!

O que é uma *race condition*?

- ✓ Race conditions
  - falha na qual o resultado produzido pela execução está inesperadamente dependente da sequência cronológica de eventos
  - podem ocorrer na programação concorrente (processos, *multithread*) e na programação distribuída
  - devem-se ao acesso não sincronizado a recursos partilhados
  - são difíceis de detetar dado o seu carácter transitório
    - Problemas apenas surgem quando existe uma(s) determinada(s) sequência(s) de eventos
      - **Ex:** T1 interrompido, T2 executa, T1 termina execução
      - Noutras sequências (T1 executa totalmente, T2 executa totalmente), tudo corre sem problemas...
    - Heisenbug vs. Bohrbug

Race condition

- ✓ Definição
  - *"Anomalous behavior due to unexpected critical dependence on the relative timing of events"*
    - [FOLDOC] Free On-Line Dictionary of Computing, http://foldoc.org/race%20condition

Solução p/ *race condition*: "exclusão mútua"

```
int Adiciona(int add){
  G_Total += add;
}
```

- A função adiciona tem que ser executada de forma "atómica" (indivisível)
  - Em "exclusão mútua"
- A função representa uma **secção crítica**
- Tal como está, a função **Adiciona** não é ***thread-safe*** nem **reentrante**

MODIFICADOR STATIC (LINGUAGEM C)

Modificador "static" no C (#1)

- ✓ A linguagem C suporta o modificador "**static**"
- ✓ Pode ser empregue em vários contextos
  - Variáveis globais do tipo *static*
  - Variáveis locais do tipo *static*
  - Funções do tipo *static*

- ✓ Uso da palavra "static" na linguagem C **não** tem o sentido usual de "estático"

Modificador "static" no C (#2)

- ✓ Variáveis têm duas caraterísticas essenciais
  - i) Duração; ii) Visibilidade
- ✓ Variáveis globais do tipo *static*
  - Duração: a do processo/aplicação
  - Visibilidade: a do ficheiro .c onde estão definidas.
    - **Não** podem ser referenciadas fora desse ficheiro.
- ✓ Exemplo – ficheiro "a.c"

```
#include <stdio.h>
// Variável apenas pode ser referenciada no "a.c"
static int count = 0;  // Variável global estática
```

Modificador "static" no C (#3)

- ✓ Variáveis locais do tipo *static*
  - Longevidade: a do processo/aplicação
  - Visibilidade: a do bloco onde estão declaradas
    - **Não** são visíveis fora desse bloco
- ✓ Exemplo

```c
#include <stdio.h>
void acrescenta(void) {
        // Variável local static
        static int local_count = 0;
        local_count++;
        printf("Contagem local: %d\n",local_count);
}
int main(void) {
        acrescenta();
        acrescenta();
        acrescenta();
        return 0;
}
```

Iniciada somente uma vez, aquando do arranque do processo

```
Contagem local: 1
Contagem local: 2
Contagem local: 3
```

Modificador "static" no C (#4)

- ✓ Função declarada com o modificador *static*
  - Visibilidade limitada ao ficheiro onde se encontra definida

```c
#include <stdio.h>
// Função estática para calcular o fatorial
static int fatorial(int n) {
        resultado = 1;
        for(int i=2; i<=n; i++){
            resultado = resultado * i;
            return resultado;
        }
}
int main() {
        int resultado = fatorial(5);
        printf("O fatorial de 5 é: %d\n", resultado);
        return 0;
}
```

CÓDIGO REENTRANTE / *THREAD-SAFENESS*

Thread-safe vs reentrante (1)

- ✓ Função reentrante
  - Pode ser chamada simultaneamente por várias threads desde que cada instância da função referencie apenas dados privados de cada thread executante
    - A função não acede a dados partilhados pelo que não está sujeita a *race condition*

- ✓ Função thread-safe
  - Pode ser chamada simultaneamente por múltiplas *threads* com cada instância a referenciar dados partilhados
  - Todos os acessos aos dados partilhados são **serializados**
    - Uso de mecanismos de sincronização por parte do programador

Exemplos >>

Thread-safe vs reentrante (2)

- Função que <u>não</u> é thread safe e <u>não</u> é reentrante
- Exemplo
  - função de tratamento de uma interrupções (*interrupt*)

```
int t; /* Variavel global */
void swap(int *x, int *y){
        t = *x;
        *x = *y;
      /* E se "isr" for novamente chamado quando estiver aqui? */
        *y = t;
}
void isr(){
        int x = 1, y = 2;
        swap(&x, &y);
}
```

Thread safeness (1)

- ✓ Um bloco de código é dito **thread-safe** se funcionar corretamente durante execuções simultâneas por várias threads

- ✓ Em particular, a ***thread safeness*** deve assegurar
  - (1) múltiplas *threads* possam aceder a dados partilhados (e.g., variável global, ...)
  - (2) Os dados partilhados devem ser acedidos de forma exclusiva apenas por uma *thread* quando essa procede a modificações (escritas)

Thread safeness (2)

- ✓ Exemplo de função thread-safe

```
int diff(int x, int y){
        int delta;
        delta = y - x;
        if (delta < 0)
                delta = -delta;
        return delta;
    }
```

delta: variável local (stack)

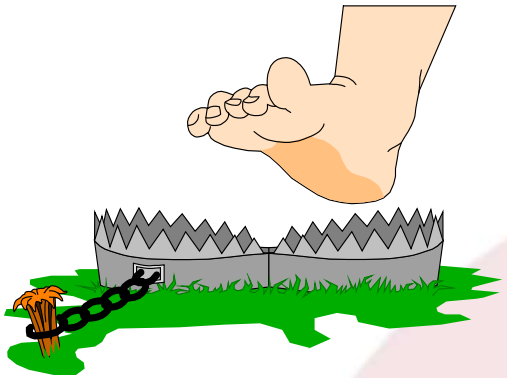
- ✓ Não acede a dados partilhados
  - Não acede a varíaveis globais, nem a variáveis estáticas
  - Apenas acede a variáveis locais
    - Cada thread tem a sua *stack*

Thread safeness (3)

✓ Exemplo de função **não** thread-safe

```
int G_total = 0;     /* Variavel global */
int diff(int x, int y){
        int delta;

        G_total + = 1;  /* acesso deve ser protegido: exclusão mútua! */
                        delta = y - x;
        if (delta < 0)
                delta = -delta;

        return delta;
}
```

G_total: = variável global



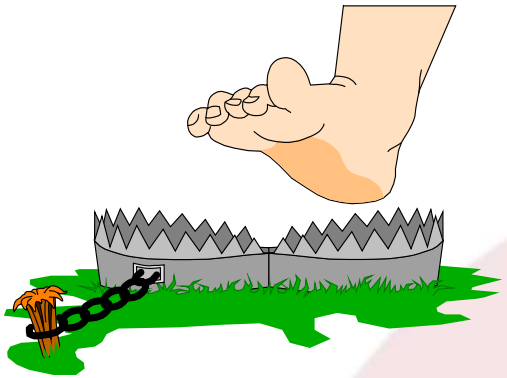
✓ Acede a variáveis globais

Thread safeness (4)

- ✓ Exemplo de função ****não**** thread-safe

static int delta: partilhada entre threads (global ao processo)

```
int diff(int x, int y){
        static int delta;/* Perigo, variável "static" */

                delta = y - x;
        if (delta < 0)
                delta = -delta;

        return delta;
    }
```



- ✓ Acede a uma variável "static"

Funções da *libc* >>

Thread safeness (5)

- ✓ Algumas funções da *libc* não são reentrantes
  - Exemplo: *ctime*(), *gmtime*(), etc.
- ✓ Versões **não** reentrantes
  - `char *ctime(const time_t *timep);`
  - `struct tm *gmtime(const time_t *timep);`
- ✓ Versões reentrantes (sufixo _r)
  - `char *ctime_r(const time_t *timep, char *buf);`
  - `struct tm *gmtime_r(const time_t *timep, struct tm *result);`

Recomendação: ler sempre o "man"

ctime()	Thread safety	MT-Unsafe race:tmbuf race:asctime env locale

EXCLUSÃO MÚTUA - MUTEXES

Mutexes (mutual exclusion)

- ✓ Como se pode proteger uma secção crítica?
  - ✓ Uso de um mecanismo de exclusão mútua
  - Variável do tipo MUTEX

    MUTEX: *mutual exclusion*

```
int Adiciona(int add){
   lock(MUTEX);
  G_Total += add;
   unlock(MUTEX);
}
```

O que é um *mutex*? >>

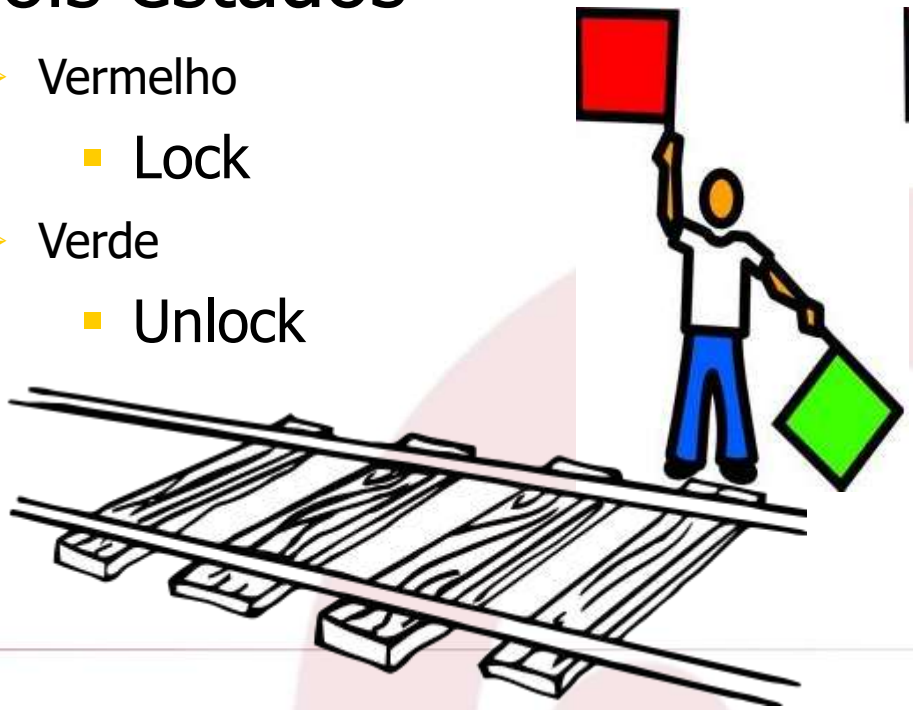
Mutexes

- ✓ **mutex** é um mecanismo de sincronização disponível para processos e threads
- ✓ Existem duas primitivas básicas para manipular *mutex*
  - **lock()**

    - permite a uma thread/processo obter o "mutex", certificando-se que apenas um fluxo de execução existe dentro da secção crítica
    - Se uma outra thread/processo chamar o lock(), fica bloqueado à espera que a outra thread/processo liberte o "mutex"
  - **unlock()**

    - assinala que uma thread/processo está a sair de uma secção crítica
    - Se existirem outras threads/processos bloqueados à espera da secção crítica, uma dessas é desbloqueada e executará (as restantes permanecem bloqueadas)

- ✓ Um mutex é também designado de semáforo binário
- ✓ Dois estados
  - ➢ Vermelho
    - ▪ Lock
  - ➢ Verde
    - ▪ Unlock



RACE CONDITION - EXEMPLO

Exemplo – race condition (1)

```
/* Increment a shared variable by C_NUM_THREADS,
 * each thread performing C_NUM_ITERS iterations.
 * Shared variable: G_shared_counter
 */
#define C_ERRO_PTHREAD_CREATE     (1)
#define C_ERRO_PTHREAD_JOIN       (2)
#define C_NUM_THREADS  (15)  /* # of threads */
/* # of times the shared value is incremented */
#define C_NUM_ITERS    (20)
int G_shared_counter;
pthread_mutex_t  G_soma_mutex;
void *soma(void *arg);
```

Exemplo – race condition (2)

```
int main(void){
        pthread_t tids[C_NUM_THREADS];
        G_shared_counter = 0;
        int i;
        pthread_mutex_init(&G_soma_mutex,NULL);
        srandom(time(NULL));
        for(i=0;i<C_NUM_THREADS;i++){
                if ((errno = pthread_create(&tids[i],NULL, soma,
                                                NULL)) != 0) {
                        fprintf(stderr,"pthread_create() "
                                "#%d failed:%s\n",i,strerror(errno));
                        exit(C_ERRO_PTHREAD_CREATE);
                }
        }
```

Vetor para armazenar os *threadsID*

Exemplo – race condition (3)

Sincronização (join)

```
        for(i=0;i<C_NUM_THREADS;i++){
                errno = pthread_join(tids[i],NULL);
                if( errno != 0 ){
                        fprintf(stderr,
                "Can't join thread #%d: %s\n", i, strerror(errno));
                        exit(C_ERRO_PTHREAD_JOIN);
                }
        }
        printf("G_shared_counter = %d (expecting %d)\n",
                G_shared_counter, C_NUM_ITERS * C_NUM_THREADS );
        pthread_mutex_destroy( &G_soma_mutex );
        return 0;
}
```

Exemplo – race condition (4)

```
void *soma(void *arg){
        (void)arg;    /* not using the 'arg' parameter */
        int i, local, num_iters = C_NUM_ITERS;
        for(i=0; i<num_iters; i++){
                /* Critical section */
//#define USE_MUTEX (1)
#ifdef USE_MUTEX
                pthread_mutex_lock(&G_soma_mutex);
#endif /* USE_MUTEX */
                        local = G_shared_counter;
                        sched_yield();
                        local = local + 1;
                        G_shared_counter = local;
#ifdef USE_MUTEX
                pthread_mutex_unlock(&G_soma_mutex);
#endif /* USE_MUTEX */
        }
        return NULL;
}
```

Exclusão mútua

Resultados

- ✓ Saída de várias execução
  - mutex **desativado** 🙄

```
G_shared_counter = 123 (expecting 300)
G_shared_counter = 131 (expecting 300)
G_shared_counter = 123 (expecting 300)
G_shared_counter = 138 (expecting 300)
G_shared_counter = 102 (expecting 300)
G_shared_counter = 136 (expecting 300)
```

  - mutex **ativado** 😀

```
G_shared_counter = 300 (expecting 300)
G_shared_counter = 300 (expecting 300)
G_shared_counter = 300 (expecting 300)
G_shared_counter = 300 (expecting 300)
…
```

SINCRONIZAÇÃO VIA SEMÁFOROS

sincronização via *semáforos* (1)

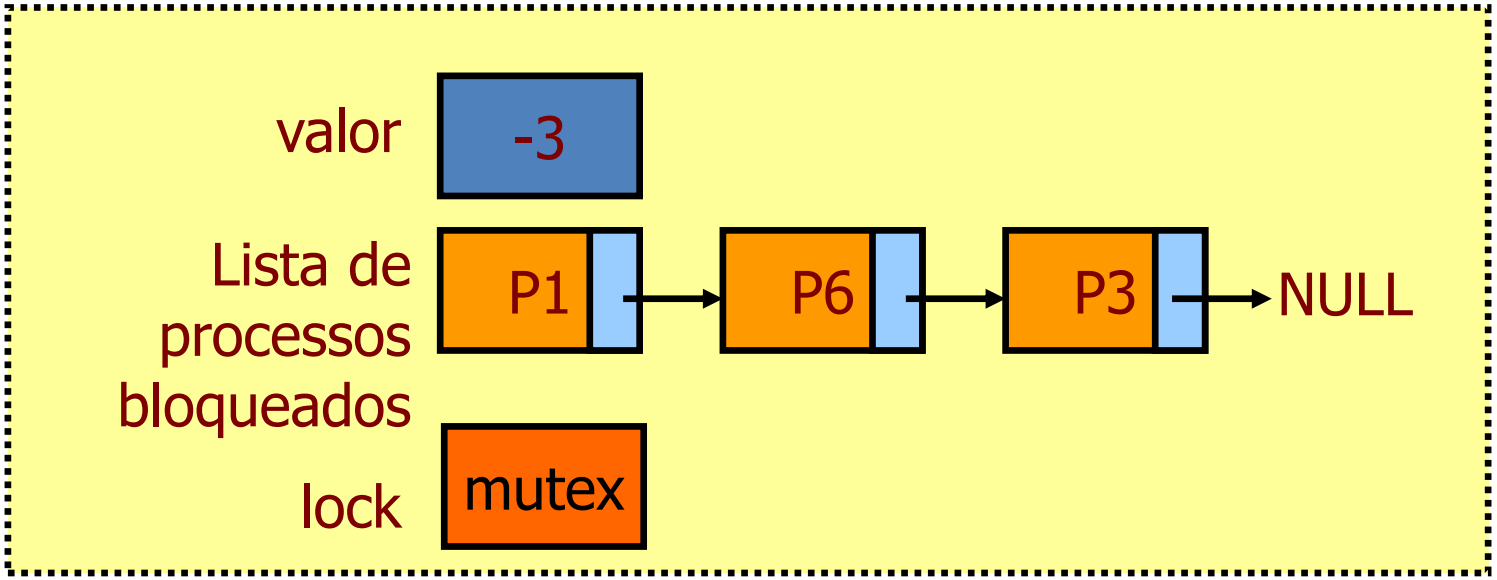
- ✓ **Um semáforo é um objeto de "sincronização"**
  - Acesso controlado a um contador (valor numérico)
  - Implementa duas operações básicas: **wait()** e **post()** (também designado por **signal()**)
  - Um semáforo é um contador de recursos
- ✓ **wait()**
  - Se o semáforo está com valor positivo, o valor é decrementado e a thread/processo prossegue
  - Se o semáforo não está como valor positivo, a thread/processo chamante é bloqueado

post / signal >>

sincronização via *semáforos* (2)

- **post()/signal()**
  - Incrementa o valor do semáforo
  - Se existiam threads/processos bloqueados no semáforo, desbloqueia uma das threads/processos
- Notas
  - Semáforos são empregues para contar "elementos" (recursos)
  - Um processo bloqueado num semáforo não consome CPU!
    - Processo está no estado *bloqueado*

Anatomia de um semáforo



Um semáforo

```
struct semaphore {
    spinlock_t lock;         /* lock para exclusão mútua de "count" */
    int count;                    /* Contador do semáforo */
    struct list_head wait_list; /* fila de espera processos bloqueados no semaf. */
    };
```

Fonte: http://lwn.net/Articles/273731/

Interfaces para semáforos (1)

- ✓ Semáforos norma POSIX – API
  - Simples de usar
  - sem_init(), sem_close(), sem_wait(), sem_post(),

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

  - Também funciona com *threads*

Interfaces para semáforos (2)

- Linguagens baseadas em linguagens interpretadas em máquinas virtuais
- Java 

  - Tem monitores (cada classe que declara *synchronized*)
  - Também têm a classe
    - **java.util.concurrent.Semaphore**
- .NET 

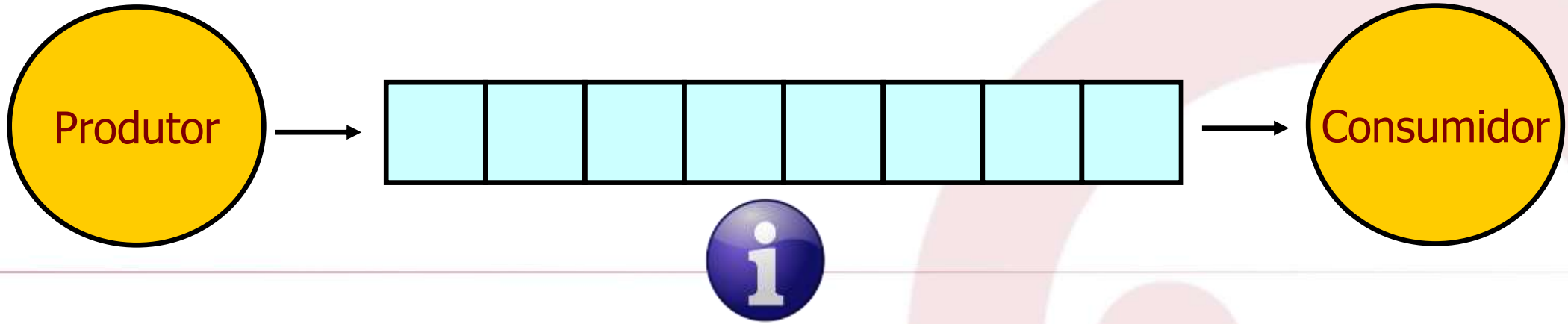
  - Também tem monitores
    - E uma classe…
    - **System.Threading.Semaphore**
- **O que é um monitor?**
  - Estrutura de sincronização usada para garantir acesso exclusivo a recursos compartilhados dentro de uma aplicação com múltiplas threads.

PROBLEMAS CLÁSSICOS DE CONCORRÊNCIA

PROBLEMA PRODUTOR/CONSUMIDOR

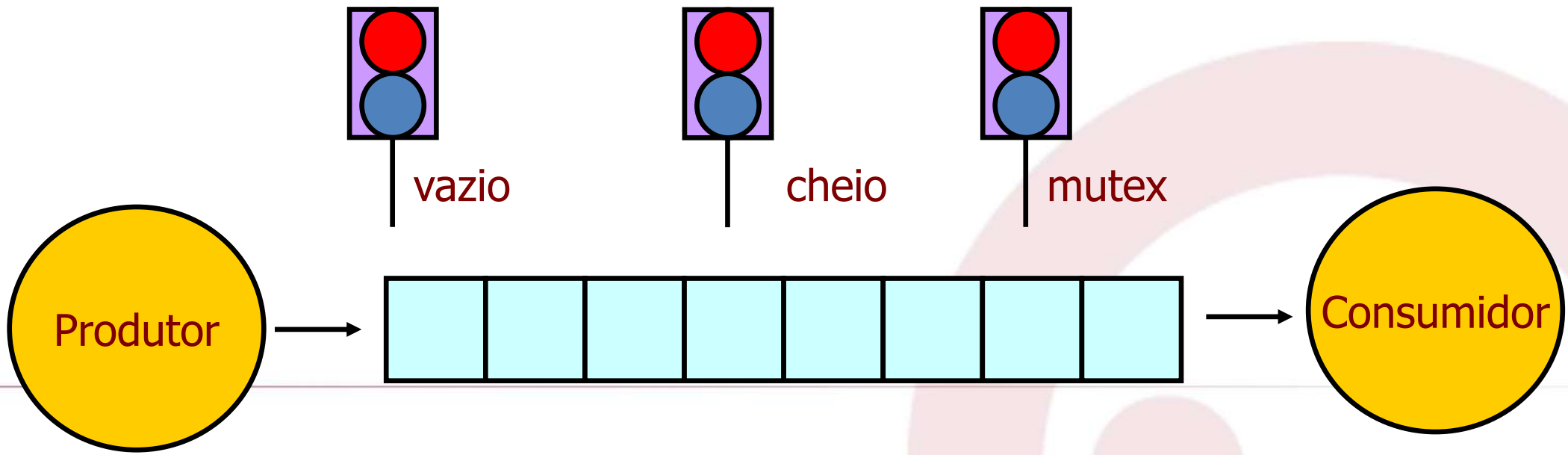
Problema produtor/consumidor

- ✓ Um "produtor" coloca elementos numa zona de memória finita (e.g. vetor)
  - Se o vetor estiver cheio, o produtor bloqueia até que haja espaço livre
- ✓ O consumidor retira ("consome") elementos
  - Se o vetor estiver vazio, o consumidor bloqueia até que algum elemento seja produzido

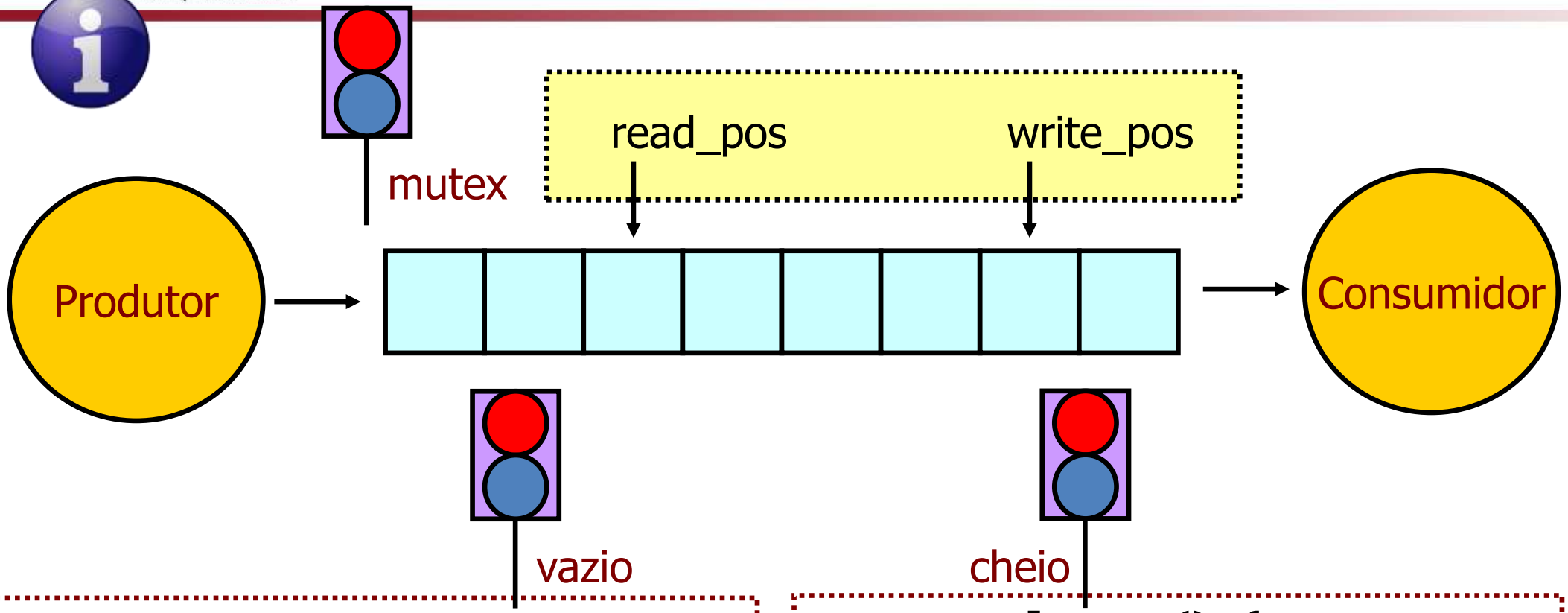


Problema "clássicos" produtor/consumidor

- ✓ São necessários três semáforos para a resolução do problema
  - Um semáforo para manter a contagem do número de posições vazias
    - Semáforo "**vazio**"
  - Um semáforo para manter a contagem do número de posições preenchidas
    - Semáforo "**cheio**"
  - Um semáforo para acesso em exclusão mútua ao vetor
    - Semáforo binário "**mutex**"



Produtor/consumidor – solução básica




```
produz_element(e) {
   sem_wait(vazio);
   sem_wait(mutex);
   buf[write_pos] = e;
   write_pos = (write_pos+1) %
N;
   sem_post(mutex);
   sem_post(cheio);
}
```

```
consome_element() {
   sem_wait(cheio);
   sem_wait(mutex);
   e = buf[read_pos];
   read_pos = (read_pos+1) % N;
   sem_post(mutex);
   sem_post(vazio);
   return e;
}
```

Ciclo principal

```
void producer() {
  for (int i=TOTAL_VALUES; i>0; i--) {
    printf("[PRODUCER] Writing %d\n", i);
    put_element(i);
  }
}
void consumer() {
  for (int i=0; i<TOTAL_VALUES; i++) {
    int e = get_element();
    printf("[CONSUMER] Retrieved %d\n", e);
    sleep(1);
  }
  terminate();
}
void main(int argc, char* argv[]) {
  init();

  if (fork() == 0) {
    producer();
    exit(0);
  }
  else {
    consumer();
    exit(0);
  }
}
```

Implementação com semáforos

put_element() e get_element()

```
void put_element(int e) {
  sem_wait(sem, EMPTY);
  sem_wait(sem, MUTEX);
  buf[write_pos] = e;
  write_pos = (write_pos+1) % N;
  sem_post(sem, MUTEX);
  sem_post(sem, FULL);
}
int get_element() {
  sem_wait(sem, FULL);
  sem_wait(sem, MUTEX);
  int e = buf[read_pos];
  read_pos = (read_pos+1) % N;
  sem_post(sem, MUTEX);
  sem_post(sem, EMPTY);
  return e;
}
```

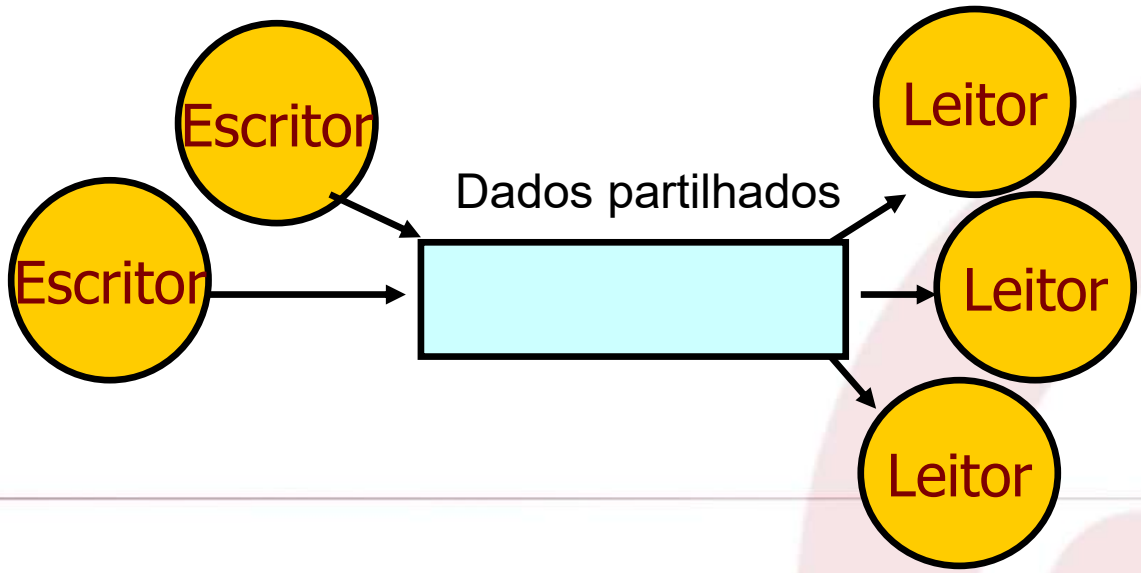
init() e *terminate()*

```
int sem, shmid;
int write_pos, read_pos;
int* buf;
void init() {
  sem = sem_get(3, 0);
  sem_setvalue(sem, EMPTY, N);        // N = número de posições no vetor
  sem_setvalue(sem, FULL,  0);
  sem_setvalue(sem, MUTEX, 1);

  write_pos = read_pos = 0;

  shmid = shmget(IPC_PRIVATE, N*sizeof(int), IPC_CREAT|0700);
  buf = (int*) shmat(shmid, NULL, 0);
}

void terminate() {
  sem_close(sem);
  shmctl(shmid, IPC_RMID, NULL);
}
```

LEITORES/ESCRITORES

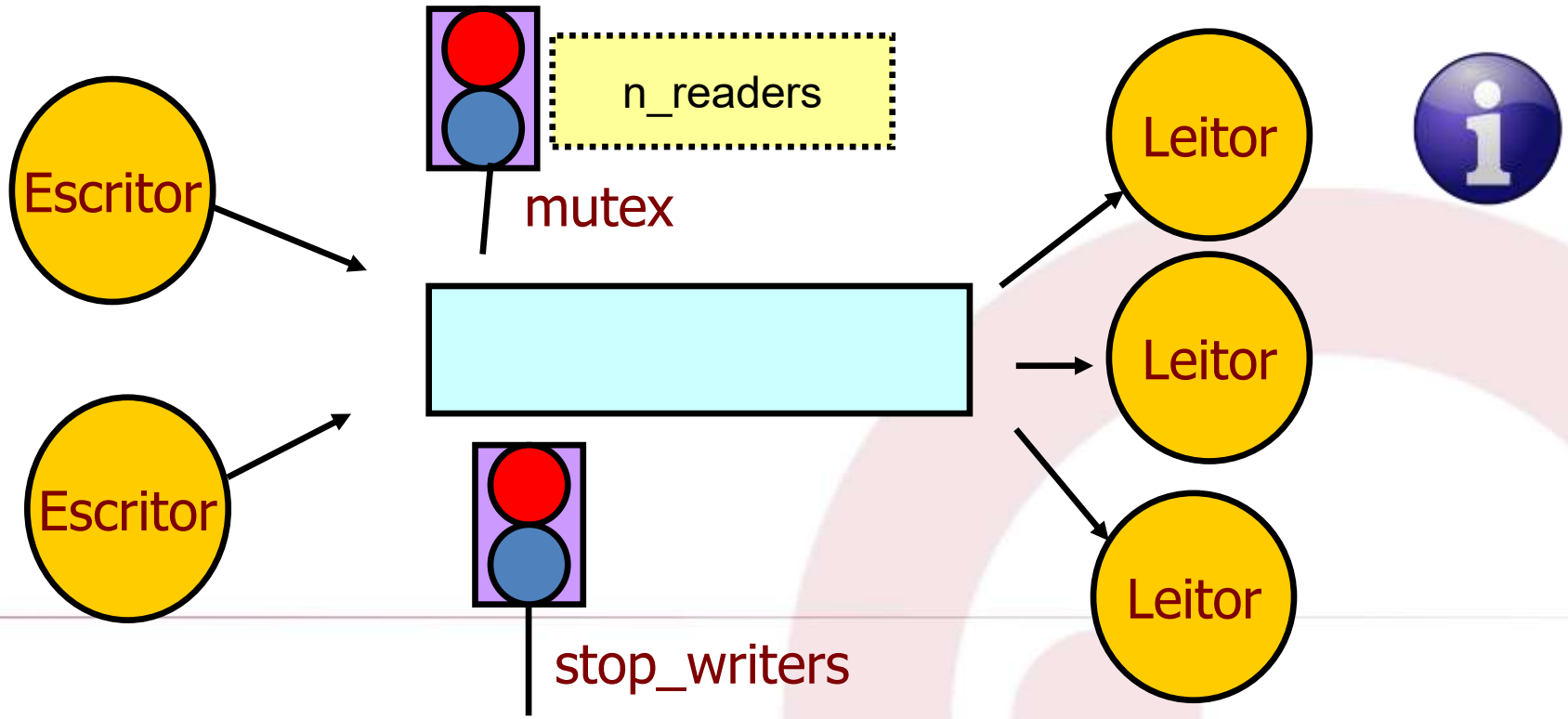
Leitores/escritores – problema clássico

- ✓ Escritores: processos actualizam dados partilhados
- ✓ Leitores: acedem aos dados partilhados
  - Mais do que um processo leitor deve poder aceder simultaneamente aos dados partilhados
- ✓ Em que é que este problema difere do "produtor/consumidor"?
  - Vários escritores/leitores com acesso simultâneo dos leitores
- ✓ Porque não usar um simples "mutex"?



Leitores/escritores (2)

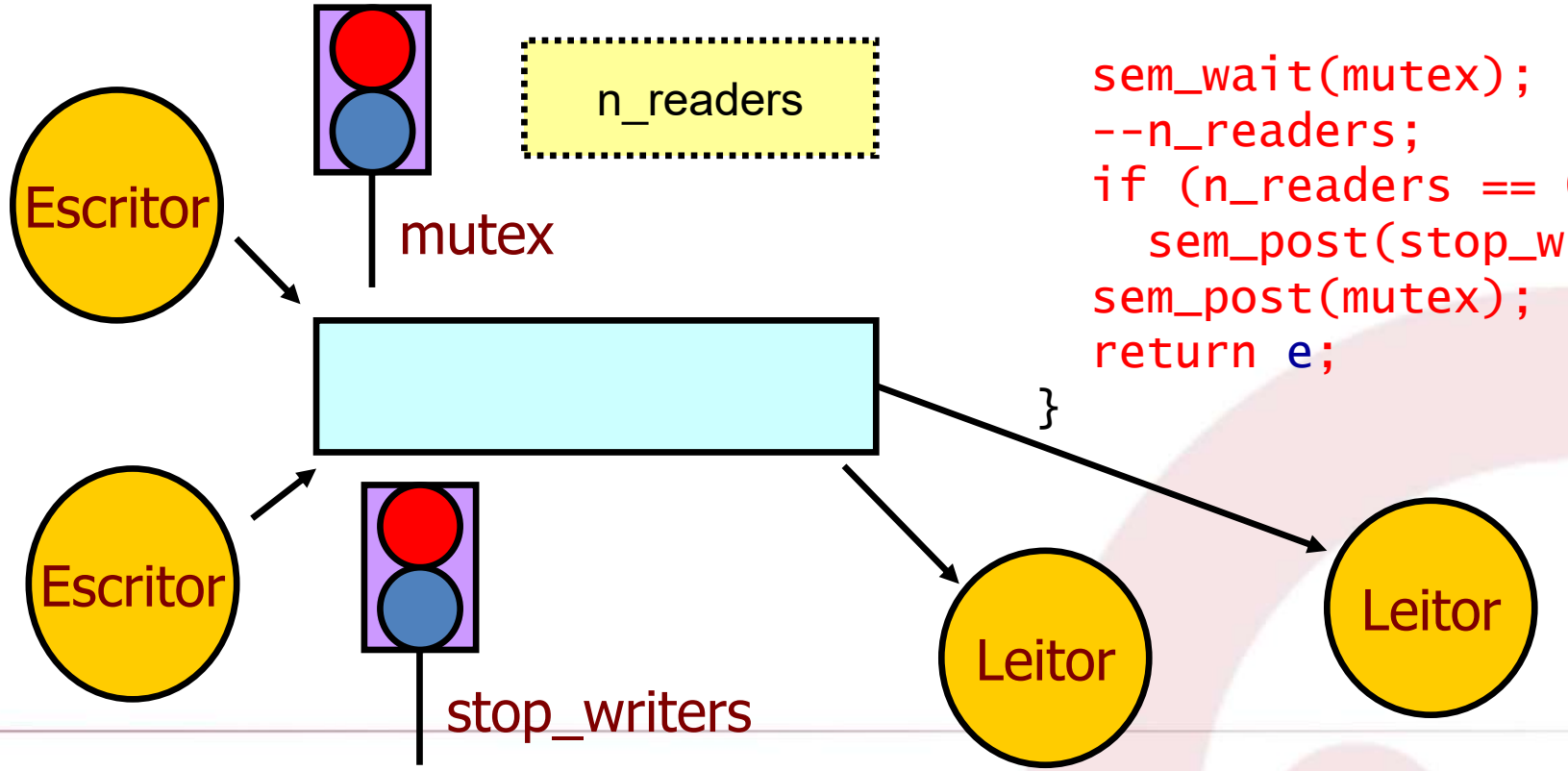
- ✓ São necessários dois semáforos
  - Um de controlo aos "escritores" (*stop_writers*)
    - Escritores em estado de espera quando já existam leitores
    - Garantir exclusão mútua quando um escritor está a atualizar os dados
  - Um para proporcionar exclusão mútua à variável partilhada "contador de leitores" (*n_readers*)



Algoritmo – prioridade aos leitores

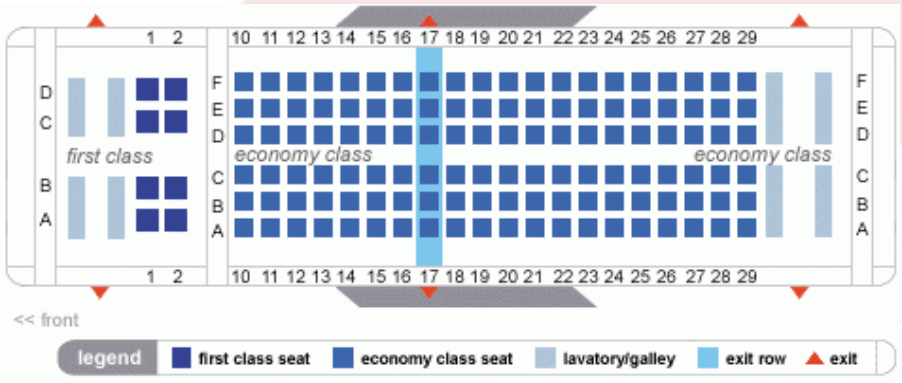
```
/* Escritor */
write(e) {
   sem_wait(stop_writers);
   buffer = e;
   sem_post(stop_writers);
}
```

```
read() {
   sem_wait(mutex);
   ++n_readers;
   if (n_readers == 1)
      sem_wait(stop_writers);
   sem_post(mutex);

   e = buffer;

   sem_wait(mutex);
   --n_readers;
   if (n_readers == 0)
      sem_post(stop_writers);
   sem_post(mutex);
   return e;
}
```



Exemplos de "leitores/escritores"…

- ✓ Algoritmo fundamental em todos os sistemas de base de dados
  - Leitura **concorrente** de dados + atualização dos dados
- ✓ Exemplos
  - …depósitos simultâneos numa conta bancária (e.g., conta de uma cadeia de supermercados, com clientes a pagarem nas caixas com multibanco); simultaneamente, outras agências ou caixas multibancos podem estar a ler (*cadeia de supermercados a efetuar pagamentos, …*)
  - …reserva via internet de transportes
    - Marcação de lugar num avião
    - Compra e marcação de lugar na rede expresso/comboio
    - …



LOCKS READER-WRITER COM PTHREADS

pthread *reader-writer* (#1)

- A norma pthread suporta *locks* reader-writer
  - Designados por 'rwlock'
  - Um *lock* leitor-escritor permite que várias threads estejam a ler um mesmo recurso simultaneamente, mas, num dado momento, apenas uma thread pode escrever no recurso.
  - Apropriado para cenários em que as operações de leitura são frequentes, mas as operações de escrita são mais raras
  - **Semântica:** Vários leitores podem adquirir o *lock* simultaneamente, mas apenas um escritor pode manter o *lock*
  - **Prioridade para escritores:** Se um escritor estiver à espera do bloqueio, irá adquiri-lo assim que possível, mesmo que existam leitores em espera.
  - **Imparcialidade:** as threads que estiveram à espera do *lock* durante mais tempo têm maior probabilidade de adquiri-lo primeiro.

pthread *reader-writer* (#2)

- pthread_rwlock_t
  - Tipo de dado que representa um lock read-write
- pthread_rwlock_init / pthread_rwlock_destroy
  - Inicializa/destroi um *lock rwlock*
- int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
  - Solicita o *lock rwlock* para leitura. Bloqueia até estar disponível
- int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);
  - Tenta obter o *lock rwlock* para leitura. Devolve erro se o lock não estiver disponível
- int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
  - Solicita o lock 'rwlock' para escrita. Bloqueia até o lock estar disponível
- int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);
  - Tenta obter o lock 'rwlock' para escrita. Devolve erro se o lock não estiver disponível
- int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
  - Liberta o *lock rwlock*

pthread *reader-writer* (#3)

- Exemplo simples do uso de um *lock rwlock*
- Nota
  - Por questões de espaço não foram considerados os códigos de retorno das chamadas às funções da API pthreads
  - Instalar as man pages na VM Linux
    - sudo apt install manpages-posix-dev

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_READERS (10)
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;
void *reader_func(void *arg) {
    (void)arg;
    pthread_rwlock_rdlock(&rwlock);
    // Leitura de dados partilhados
    // (...)
    pthread_rwlock_unlock(&rwlock);
    return NULL;
}
void *writer_func(void *arg) {
    (void)arg;
    pthread_rwlock_wrlock(&rwlock);
    // Escrita nos dados partilhados
    pthread_rwlock_unlock(&rwlock);
    return NULL;
}
int main(void) {
    pthread_t reader_threads[NUM_READERS];
    pthread_t writer_thread;
    for (int i = 0; i < NUM_READERS; i++) {
        pthread_create(&reader_threads[i], NULL, reader_func, NULL);
    }
    pthread_create(&writer_thread, NULL, writer_func, NULL);

    for (int i = 0; i < NUM_READERS; i++) {
        pthread_join(reader_threads[i], NULL);
    }
    pthread_join(writer_thread, NULL);
    return 0;
}
```

RACE-CONDITION EM FICHEIROS TEMPORÁRIOS

Ficheiros temporários (#1)

- ✓ Aplicações recorrem frequentemente a ficheiros temporários
  - No Linux, é empregue o diretório /tmp ou /var/tmp
    - Diretórios partilhados
  - Todos os processos podem criar ficheiros nos diretórios /tmp e /var/tmp
- ✓ Diretórios /tmp e /var/tmp tem permissões especiais
  - `drwxrwxrwt 12 root root 4096 Sep 17 00:04 /tmp`
  - `drwxrwxrwt 2 root root 4096 Sep 16 23:59 /var/tmp`
  - Permissão "t" é designada por *sticky bit*
    - Uma pasta marcada com o bit "t" apenas permite que o dono do ficheiro/da pasta/ou o root possam alterar/apagar um ficheiro
    - Ativar sticky bit numa pasta
      - chmod +t /pastaXPTO Adiciona a permissão "t"

Ficheiros temporários (#2)

- ✓ Aplicações seguras devem seguir alguns cuidados no uso de ficheiros
  - Criação de ficheiro com O_CREAT | O_EXCL e atribuir permissões apenas ao dono do ficheiro
    - Se o ficheiro que se pretende criar já existir, a tentativa de abertura do ficheiro falha
- ✓ Uso de ficheiros em diretórios temporários
  - Todos os utilizadores do sistema têm permissão de escrita em diretórios temporários (e.g., /tmp)
  - É necessário criar o ficheiro de forma atómica
    - Garantir que o ficheiro a criar não existe
    - Depois de abrir o ficheiro, deve-se usar sempre o descritor do ficheiro (e não reabrir o ficheiro)

Função **mkstemp**

- ✓ Criar ficheiro temporário evitando *race-conditions*
  - Recomenda-se o uso da função **`mkstemp`**
    - `int mkstemp(char *template);`
  - `man 3 mkstemp`
    - *The mkstemp() function generates a unique temporary filename from template, creates and opens the file, and returns an open file descriptor for the file*
    - Template deve ser um vetor de caracteres, pois é modificado pela função
      - Nome final do ficheiro é retornado na variável *template*
    - Nome a passar (pela função chamante) na variável template deve terminar por XXXXXX

mkstemp - exemplo

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

int main() {
    char template[] = "tempfile_XXXXXX";
    int fd;
    // Create a unique temporary file with mkstemp
    fd = mkstemp(template);
    if (fd == -1) {
        perror("mkstemp");
        exit(EXIT_FAILURE);
    }
    printf("Temporary file created with descriptor: %d\n", fd);
    // You can use fd to read from and write to the temporary file
    write(fd,"TESTE!", strlen("TESTE!"));
    // Close the temporary file when you're done with it
    close(fd);
    // Remove the temporary file (optional)
    //unlink(template);
    return 0;
}
```

```
-rwxrwxr-x 1 user user 16216 out 10 01:11 mkstemp
-rw-rw-r-- 1 user user   636 out 10 01:11 mkstemp.c
-rw------- 1 user user     6 out 10 01:11 tempfile_kWmHYg
```

Seis **X**

Ficheiro temporário

https://pastebin.com/DbV3Wf7a

SINCRONIZAÇÃO

Sincronização – regras básicas...(1)

- Nunca alternar *locks* diferentes!

  - Os *locks* devem ser tomados ("*wait*") pela mesma ordem por todos os processos intervenientes

  - Os *locks* devem ser libertados ("*post*") pela ordem inversa da ordem em que foram tomados

Exemplo >>

Sincronização – regras básicas…(2)

- Processo 1

```
sem_wait(A)
sem_wait(B)
// Secção crítica
sem_post(B)
sem_post(A)
```

- Processo 2

```
sem_wait(B)
sem_wait(A)
// Secção crítica
sem_post(A)
sem_post(B)
```

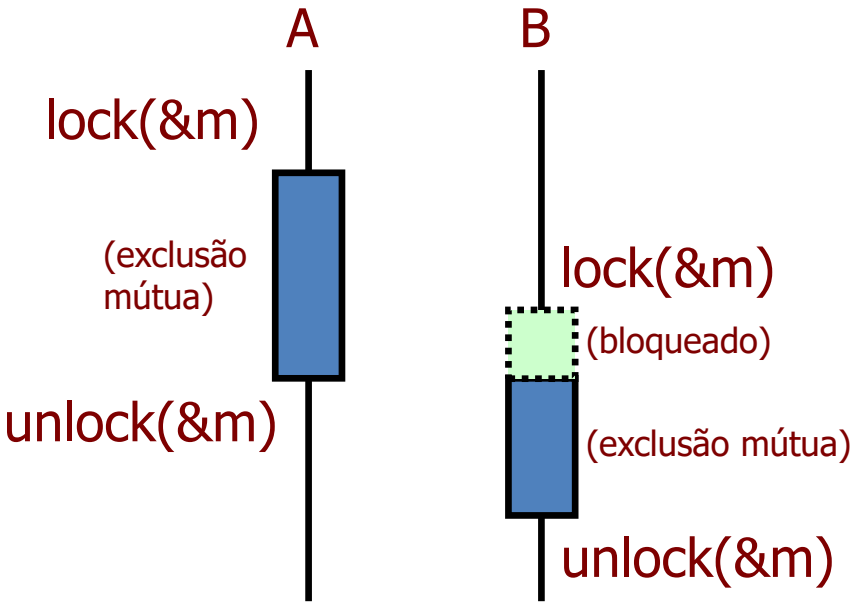
deadlock!

- ✓ O processo #1 vai ficar bloqueado à espera do recurso controlado pelo semáforo B
- ✓ O processo #2 vai ficar bloqueado à espera do recurso controlado pelo semáforo A
- ✓ Mas...
  - ➢ #1 bloqueado detém "A" e aguarda B
  - ➢ #2 bloqueado detém "B" e aguarda A

Sincronização – regras básicas...(3)

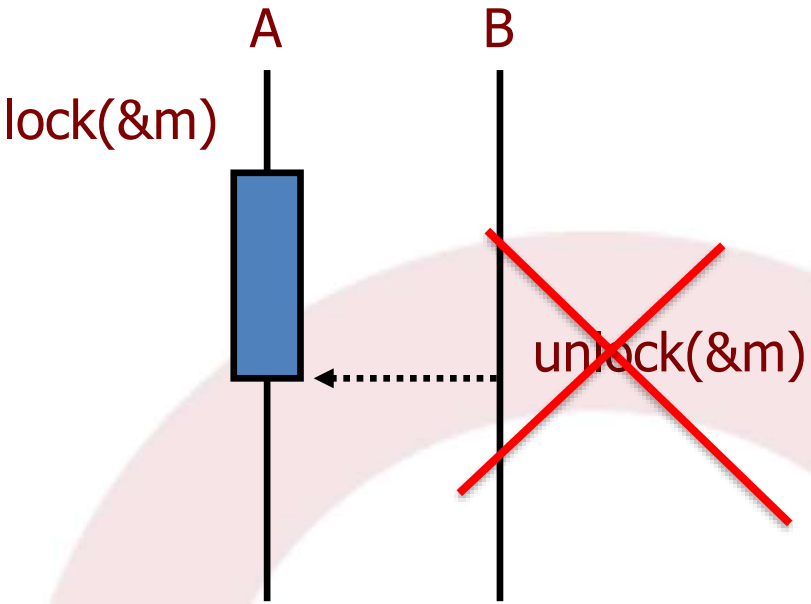
- Os "mutexes" são para implementar exclusão mútua, não para assinalar eventos entre threads.
  - Apenas a thread que possui o lock num mutex pode desbloqueá-lo (princípio da exclusão mútua)
- Como assinalar eventos entre *threads*?

  - usar variáveis de condição!



CORRETO!

Uso para exclusão mútua



INCORRETO!

Alguma terminologia

✓ Deadlock

- Quando dois ou mais processos estão bloqueados, aguardando por recursos que estão na posse de outros processos (também eles bloqueados)

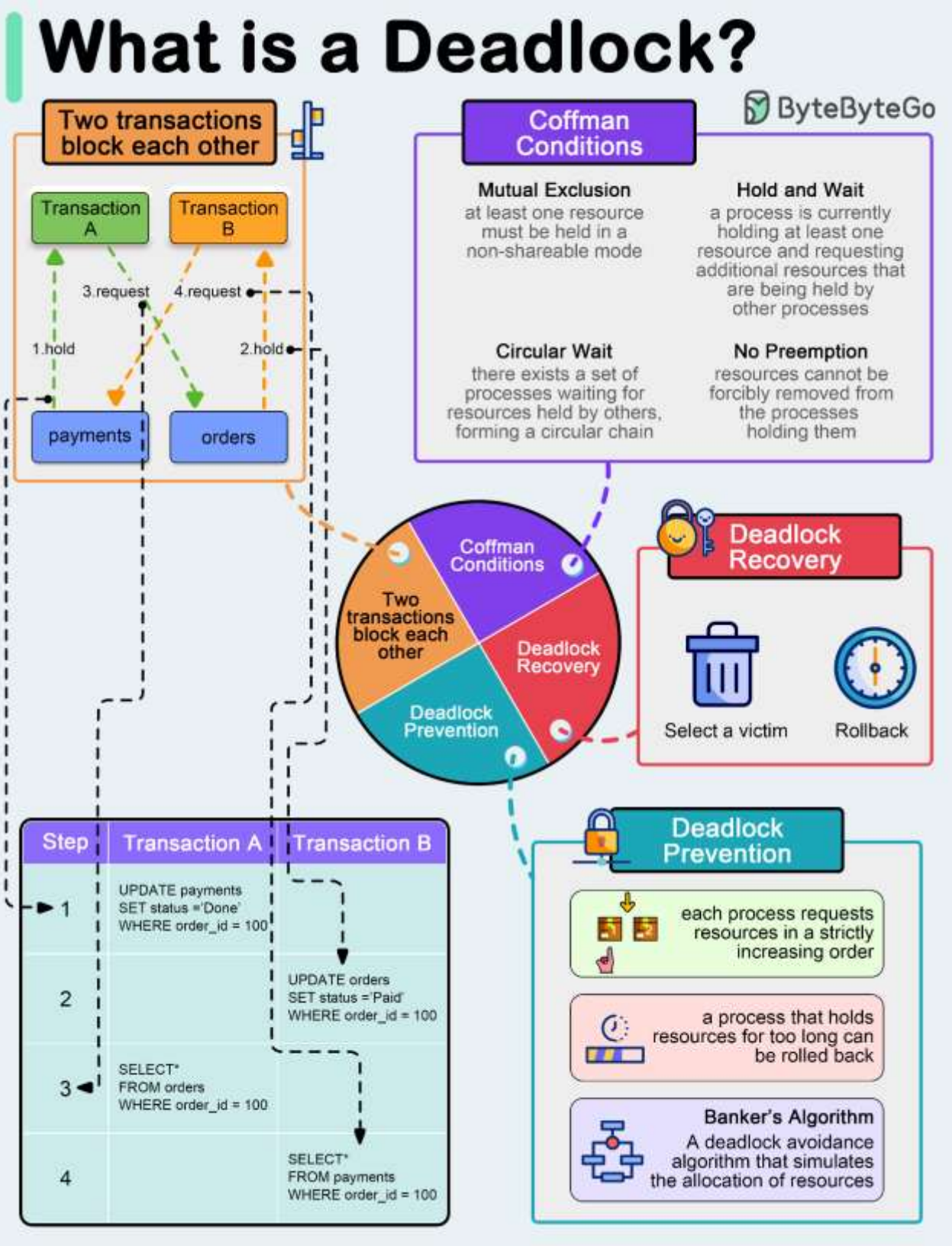
✓ Livelock

- Quando dois ou mais processos estão a processar, mas impedidos de progredir (semelhante ao deadlock, mas com a agravante de consumo de CPU)
  - similar a duas pessoas a tentarem passar por um corredor estreito

✓ Starvation (fome)

- Quando um processo não consegue aceder aos recursos que precisa para continuar

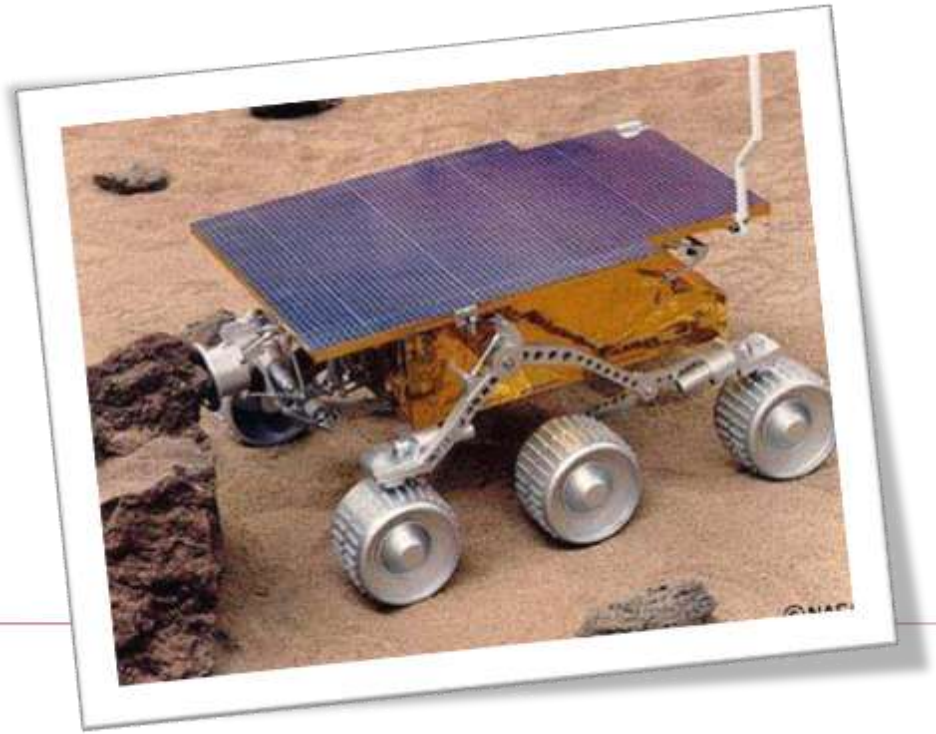
- ✓ Fonte
  - – ByteByteGo



CASO DE ESTUDO - SOJOURNER

Sincronização – inversão de prioridades

- ✓ Cuidado com os níveis de prioridades
  - ➢ Threads de baixa prioridade que possuam determinado recurso podem bloquear thread de alta prioridade...
    - Não é isso que se quer!



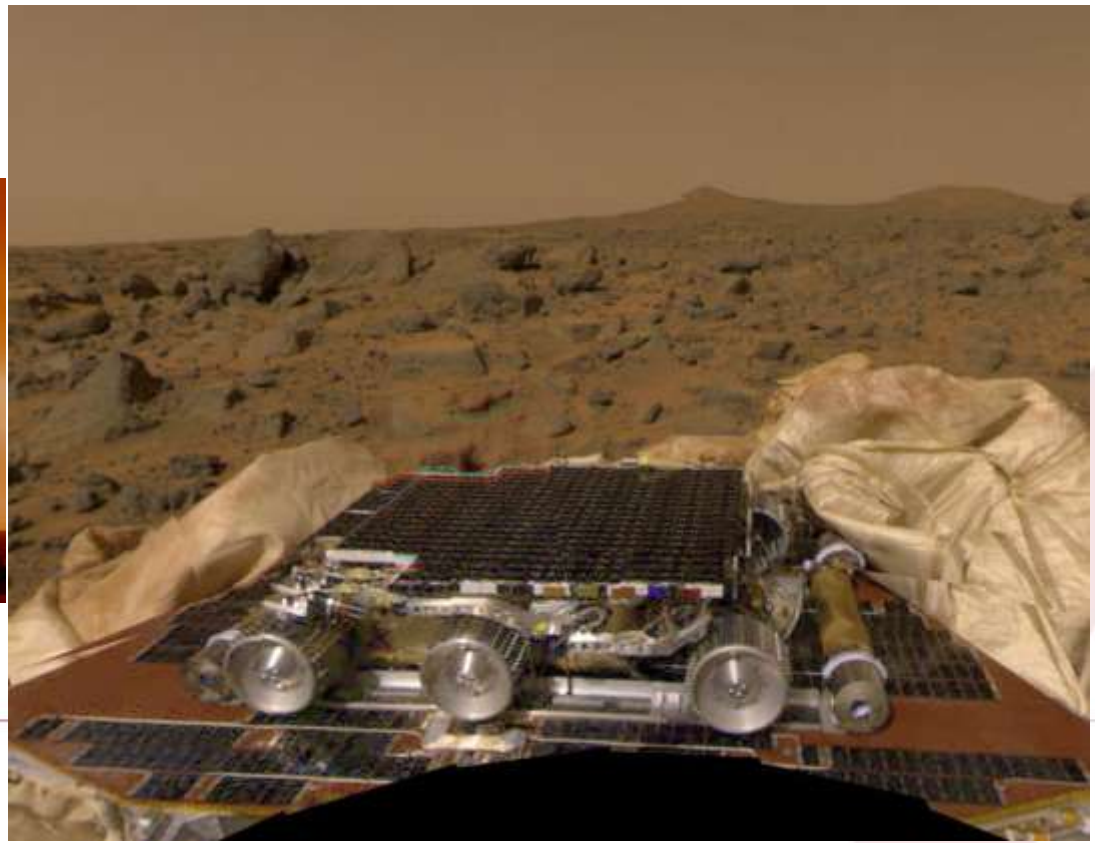
Problema no Mars PathFinder (1997)

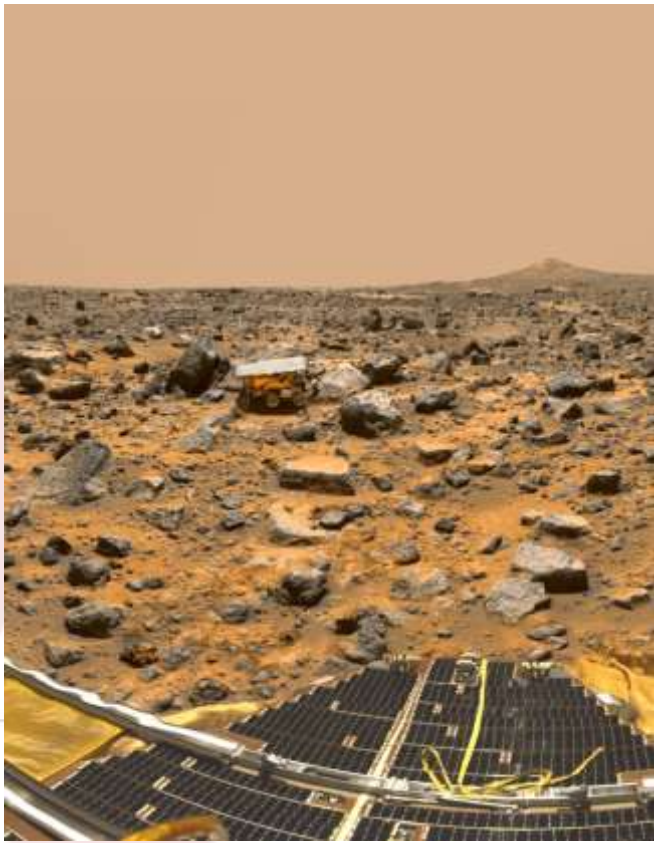
"The Mars PathFinder would run ok for a while and, after some time, it would stop and reset.
The watch-dog timer was resetting the system because of some unknown reason…"

+info: pesquisar "Sojourner priority inversion problem"

Sojourner (1997)

- ✓ The lander, formally named the Carl Sagan Memorial Station following its successful touchdown, and the rover, named after American civil rights crusader Sojourner Truth, both outlived their design lives — the lander by nearly three times, and the rover by 12 times.
- ✓ Mars Pathfinder returned more than 16,500 images from the lander and 550 images from the rover, as well as chemical analysis of rocks and soil and extensive data on winds and other weather factors.

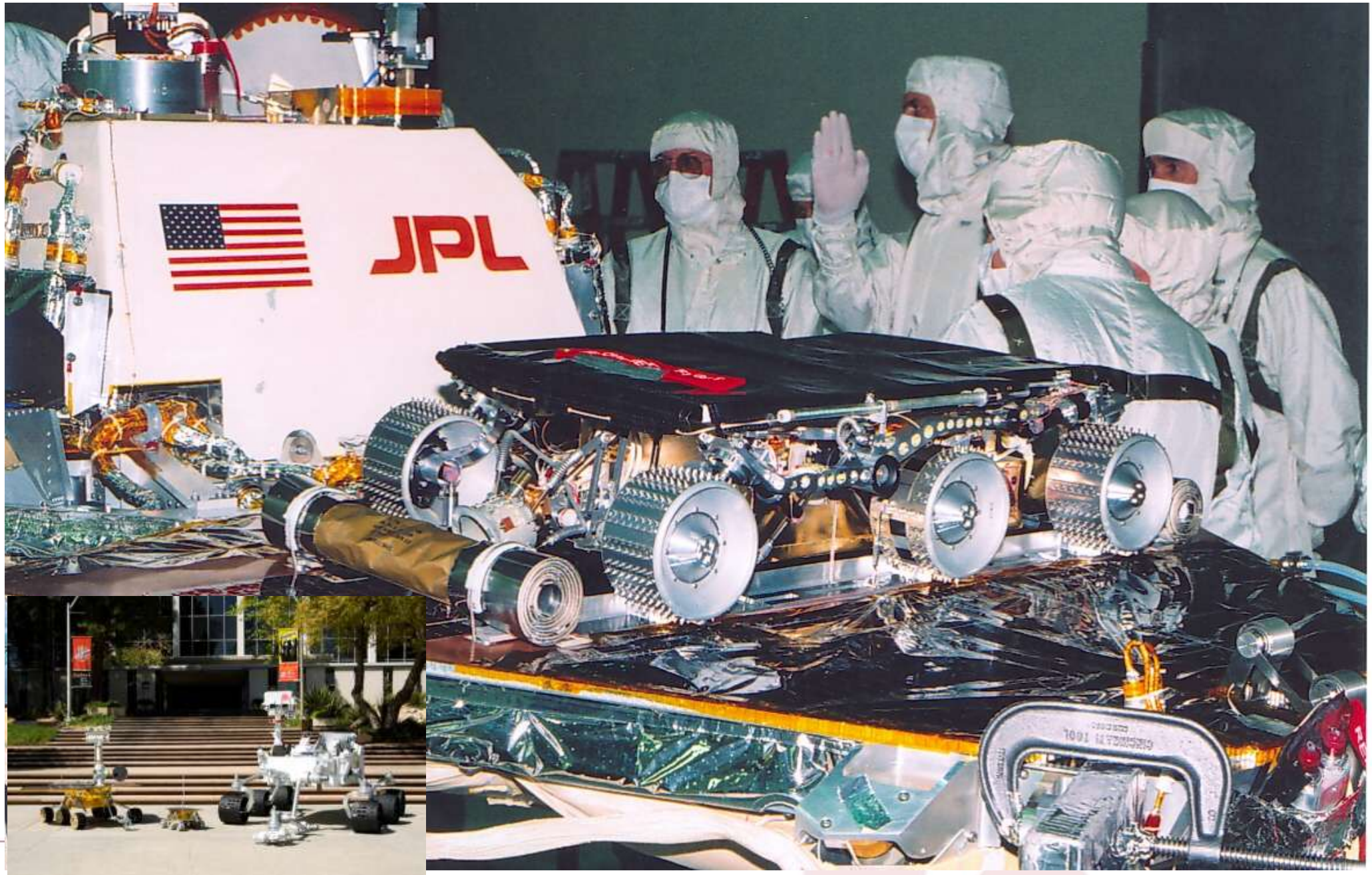




IPL
escola superior de tecnologia e gestão
instituto politécnico de leiria

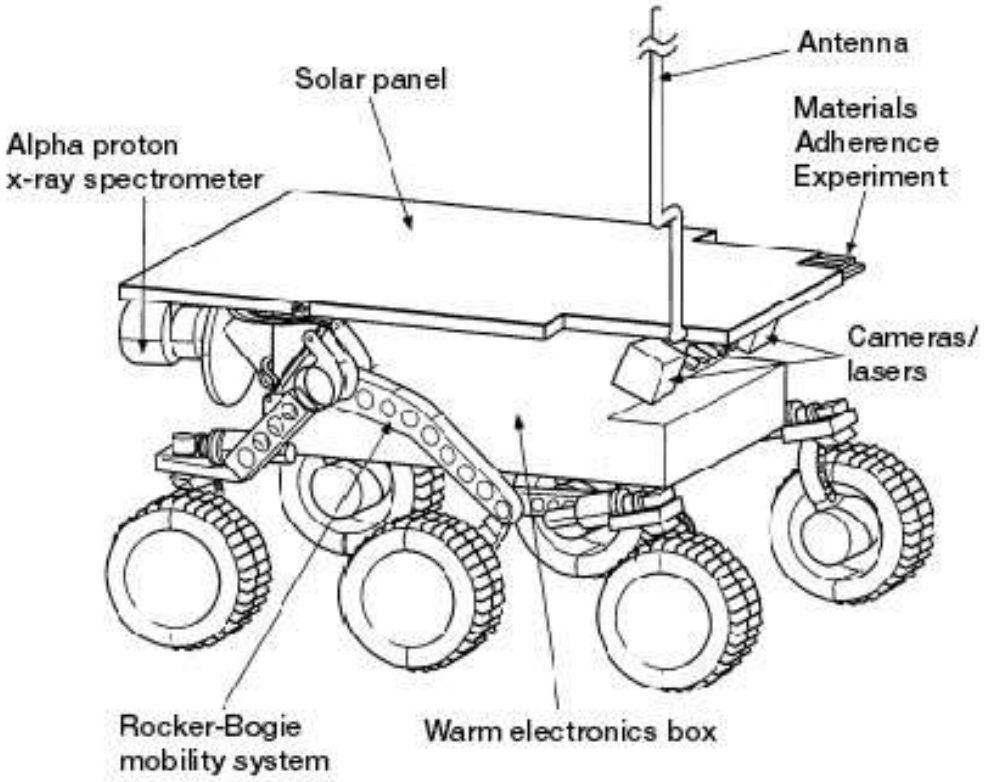
Sojourner (1997)

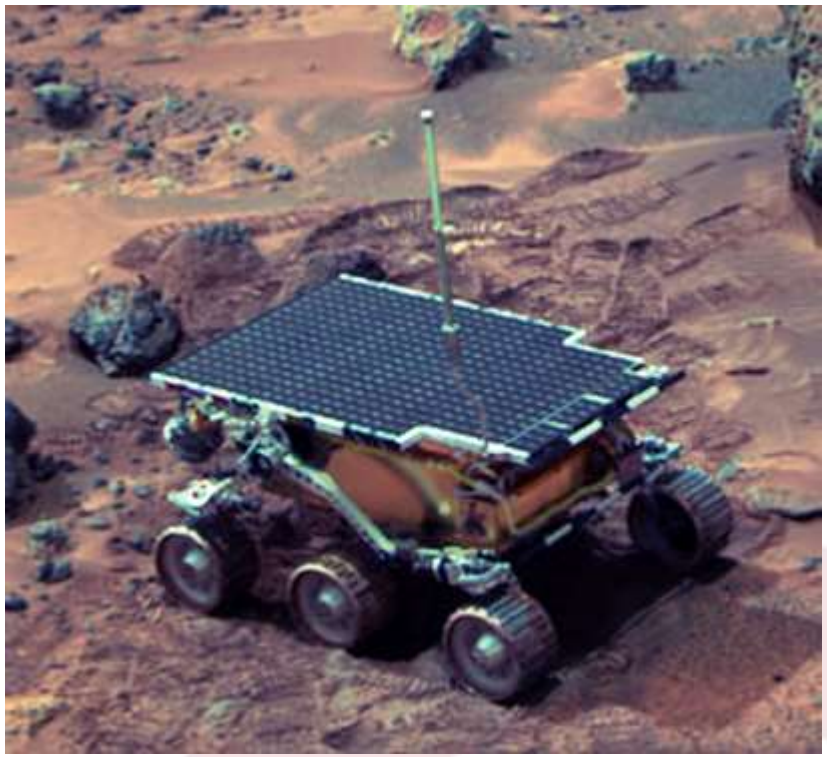
NASA



Sojourner (1997)

- ✓ Sojourner…em Marte



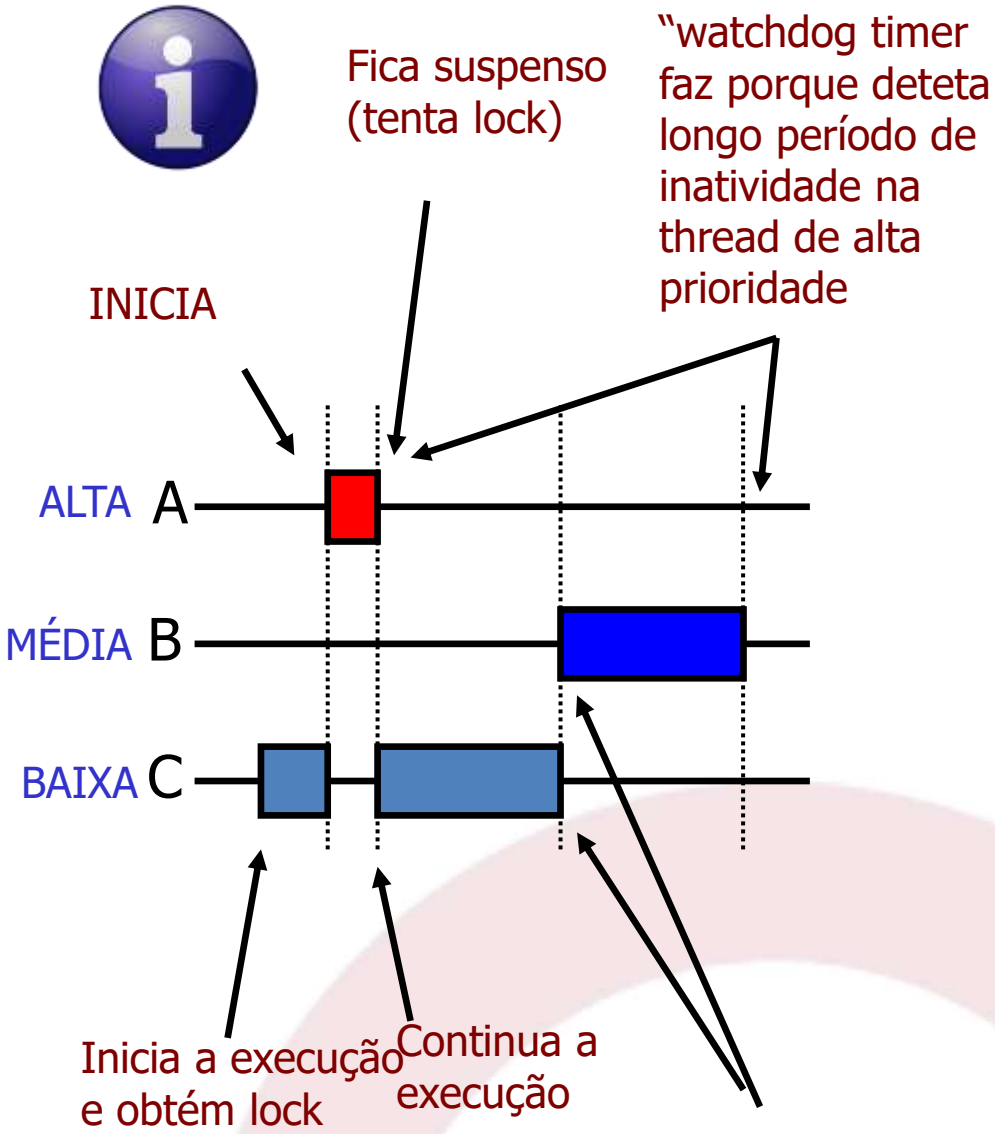


O problema do "Sojourner" (1997)

1. Thread baixa prioridade (dados meteorologia) "bloqueia" semáforo de acesso ao bus de comunicação
2. Thread alta prioridade (tarefas de comunicação) inicia execução e tenta obter o mesmo semáforo
3. Thread média prioridade executa – não precisa do semáforo (thread baixa fica sem CPU).

- Entretanto a thread de alta prioridade continua bloqueada...
  - Sistema deteta e faz reset

"Houston, we've had a problem"



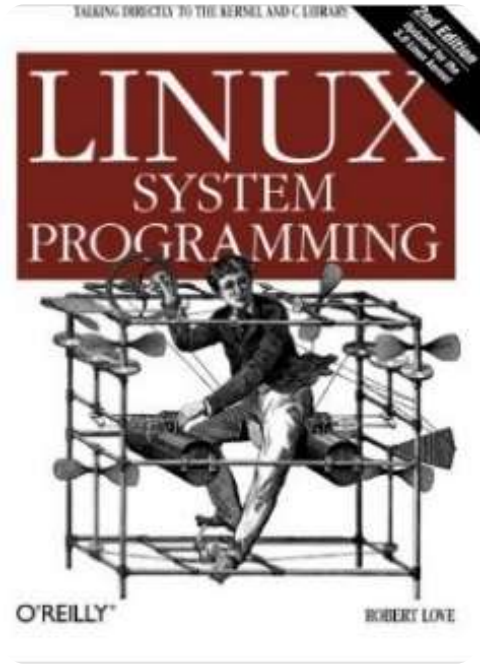
Note: *"It took three weeks to find the problem and another 18 hours to fix it; the issue turned out to be buried deep down in the VxWorks mechanics."*

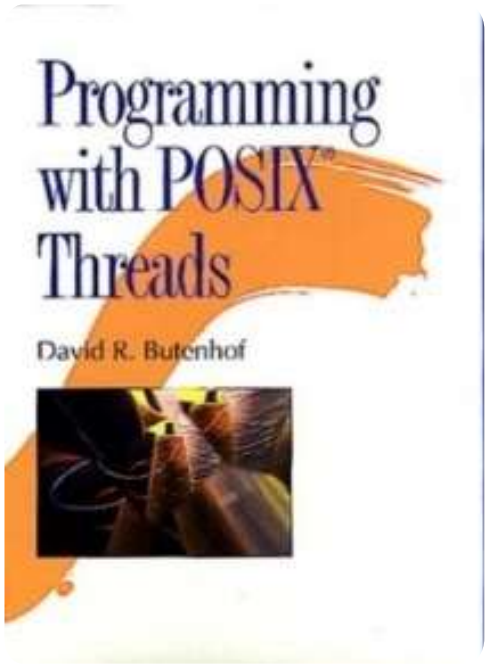
Bugs em sistemas concorrentes

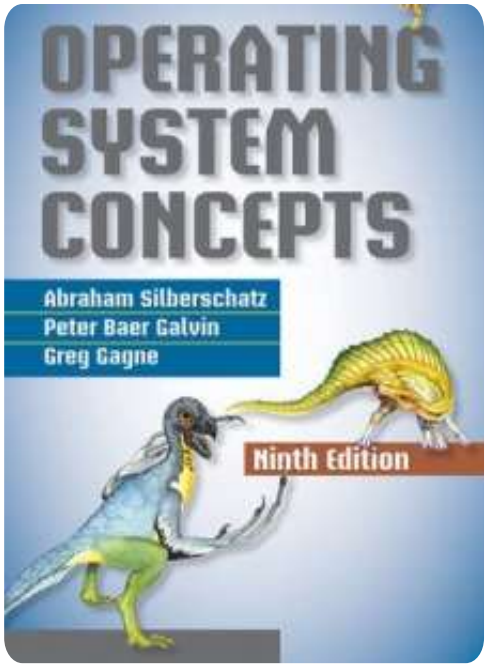
- Estudo: *Lu et al. "Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics" by Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou. ASPLOS '08, **March 2008**, Seattle, Washington.*
  - **Fonte:** "Chapter 32 – Common concurrency Problems", Arpaci-Dusseau, Remzi H., and Andrea C. Arpaci-Dusseau. Operating systems: Three easy pieces. Arpaci-Dusseau Books LLC, 2018.

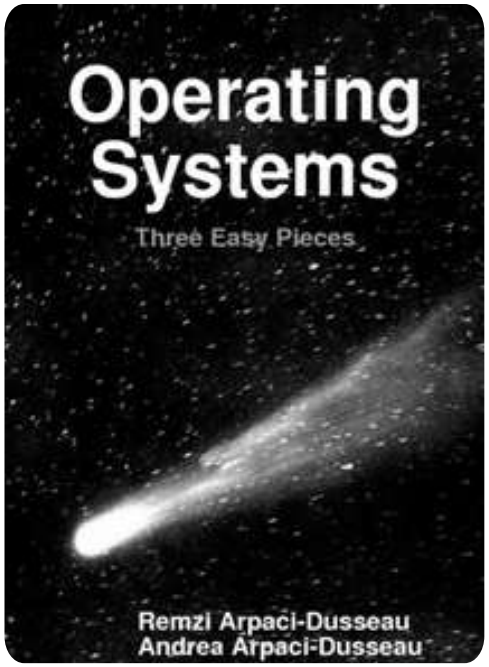
Application	What it does	Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Web Browser	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Figure 32.1: **Bugs In Modern Applications**









Bibliografia

- “Linux System Programming”, Robert Love, Cap. 7 - Threading, O’Reilly, 2ª edição, 2013.
- “Operating Systems: three easy pieces” – part 2 – concurrency, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, 2018. www.ostep.org
- "Programming with POSIX Threads", David R. Butenhof, Addison-Wesley, 1997, ISBN-13: 978-0201633924
- Operating System Concepts, 9th Edition, Abraham Silberschatz, Peter B. Galvin, Greg Gagne, 2012 – Cap. 5 (Process Synchronization) & Cap. 7 (Deadlocks)