

gdb

threads
i++
mutex
IPL++
tree
ponteiro
linked list
ciclo
gccchar *ptr;
for#include
(c) Patrício Domingues
doxygen
sockets
lock/unlock
#define
malloc

Programação Avançada

Sincronização de Sistemas Concorrentes Parte I

Patrício Domingues

- ✓ “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 uma zona de memória
 - Cenário para concorrência
 - N threads acedem à variável
 - Pelo menos uma thread escreve na variável
 - Se não existir sincronização no acesso à variável partilhada então existirá uma corrida por recurso (*race condition*)



- 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 >>



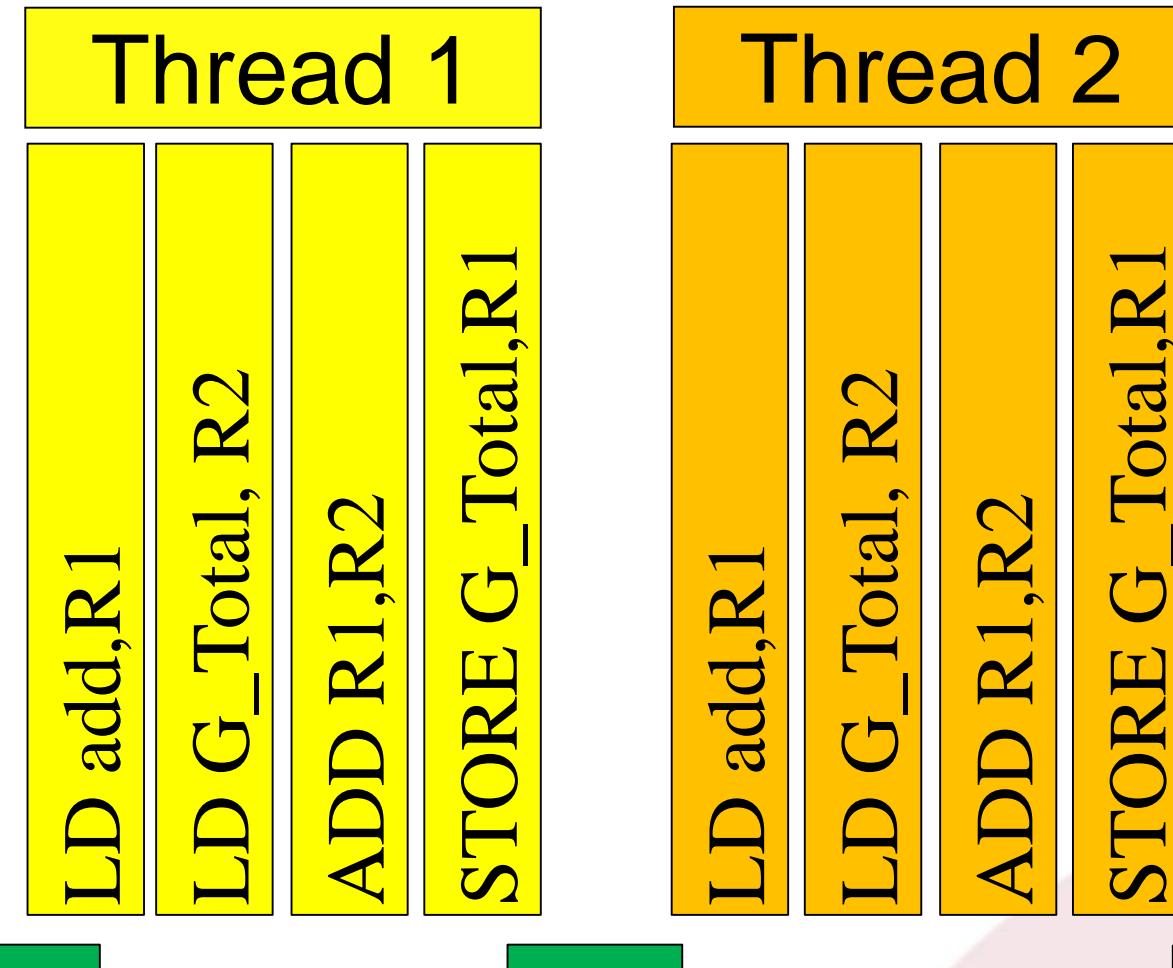
Variável global e duas threads... (2)

- ✓ 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 */
```

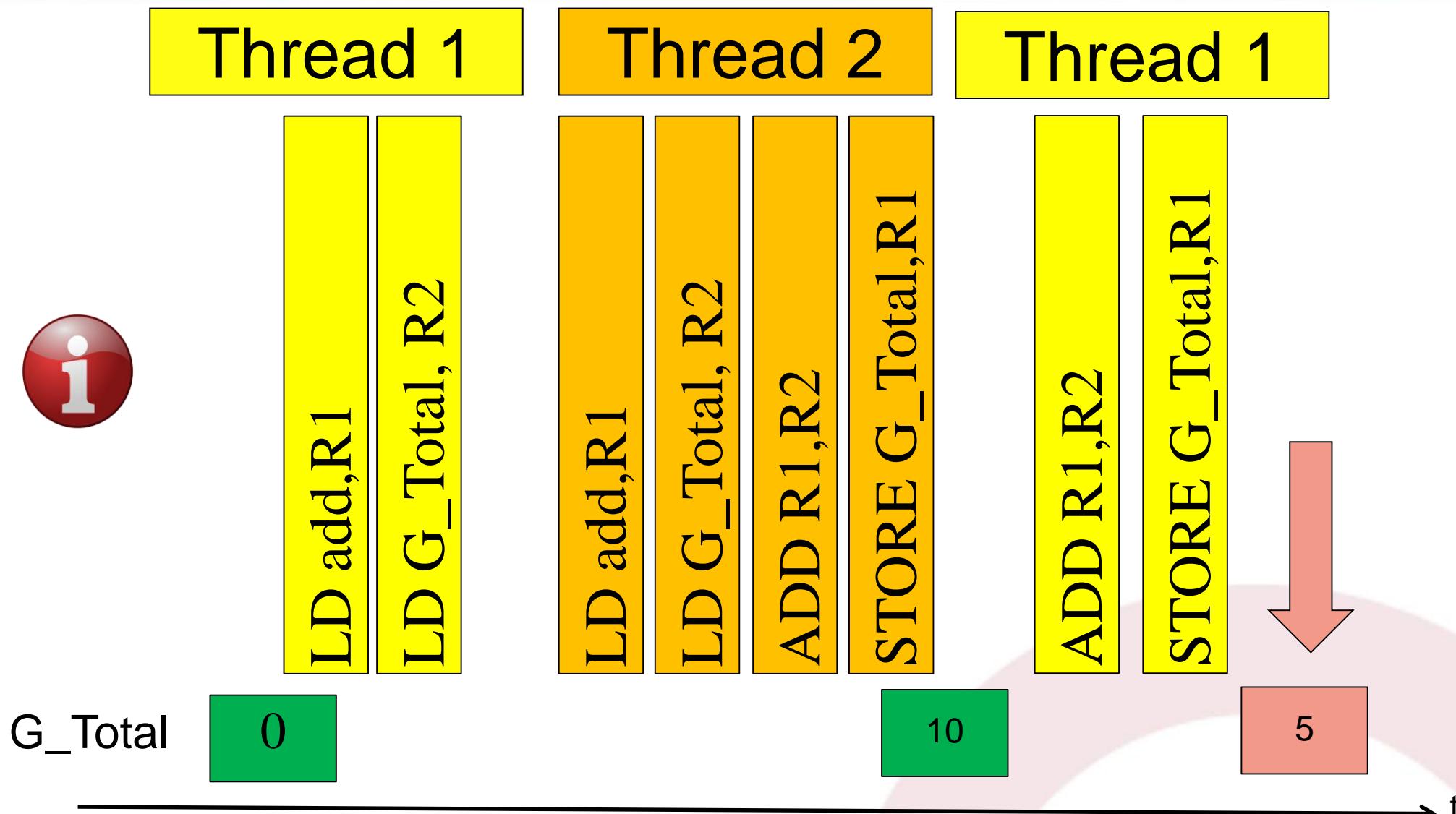
- ✓ Uma instrução “C” é mapeada para quatro instruções assembler
 - ✓ A instrução C não é atómica...

Variável global e duas threads... (3)



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

Caso 2 >>



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

Caso 3 >>



G_Total

0

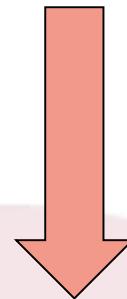
Thread 2

LD add,R1
LD G_Total, R2

Thread 1

LD add,R1
LD G_Total, R2
ADD R1,R2
STORE G_Total,R1

Thread 2

ADD R1,R2
STORE G_Total,R1

5

10

t

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

Análise >>



✓ 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!



IPL

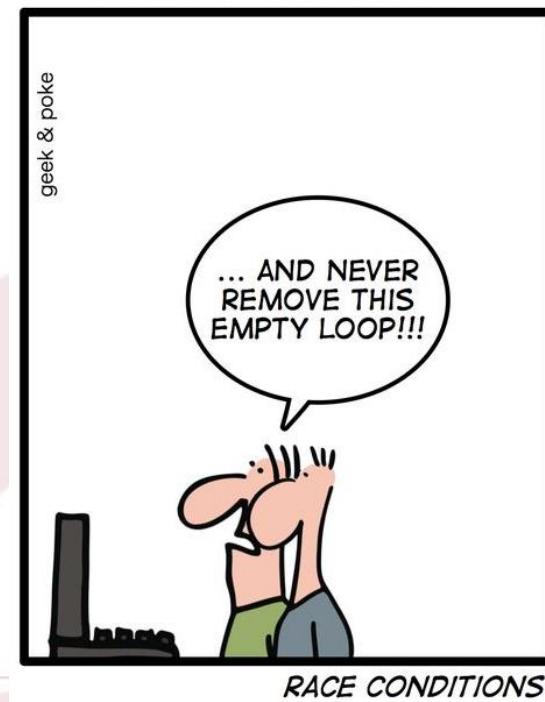
escola superior de tecnologia e gestão
Instituto Politécnico de Leiria

O que é ao certo 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

SIMPLY EXPLAINED



RACE CONDITIONS

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



✓ 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

✓ 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

[Exemplos >>](#)

Thread-safe vs reentrante (2)

- Função que não é thread safe e não é reentrante
- Exemplo
 - função de tratamento de um *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;  
}
```



- ✓ Não acede a dados partilhados
 - Não acede a variáveis 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;
```

```
}
```



- ✓ Acede a variáveis globais

Thread safeness (4)

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

```
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”

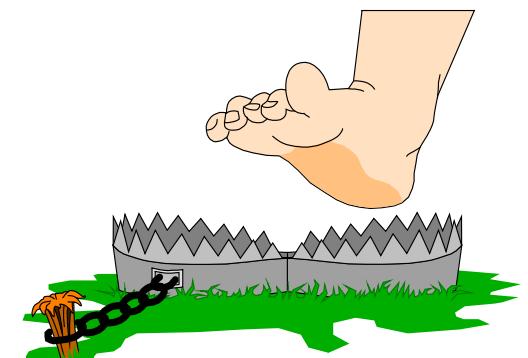
Funções da *libc* >>

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”

<code>ctime()</code>	Thread safety	MT-Unsafe race:tmbuf race:asctime env locale
----------------------	---------------	---

Mutexes (*mutual exclusion*)

- ✓ Como se pode proteger uma secção crítica?
 - ✓ Uso de um mecanismo de exclusão mútua
 - ✓ Variável **MUTEX**
 - ✓ **MUTEX**: mutual exclusion

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

O que é um *mutex*? >>

- ✓ **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()**

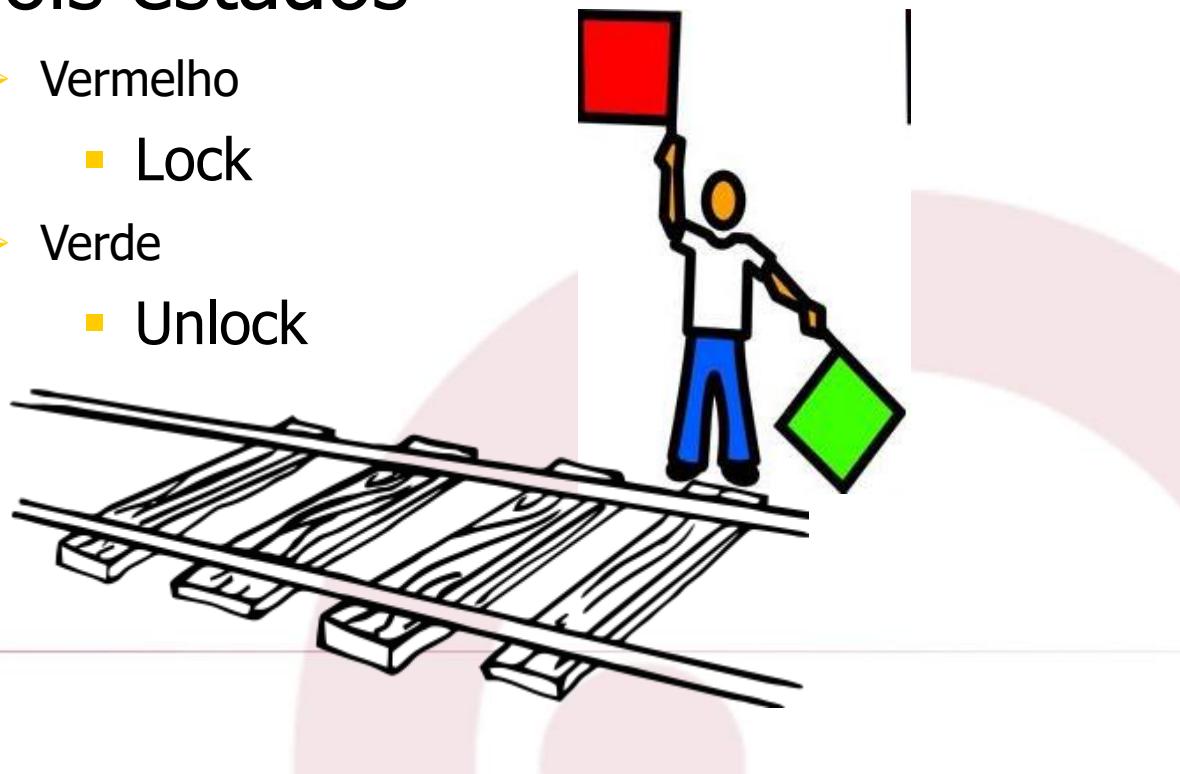


- assinala que uma thread/processo está a sair de uma secção crítica
- Se existirem outras threads/processos bloqueados à espera da secção crítica, uma dessas é desbloqueada e executará (as restantes permanecem bloqueadas)



- ✓ Um mutex é também designado de semáforo binário
- ✓ Dois estados

- Vermelho
 - Lock
- Verde
 - Unlock



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);
    srand(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);
        }
    }
}
```

Exemplo – race condition (3)

```
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;
#endif USE_MUTEX
        pthread_mutex_unlock(&G_soma_mutex);
#endif /* USE_MUTEX */
    }
    return NULL;
}
```

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)
```

...



✓ Um semáforo é um objeto de “sincronização”

- Acesso controlado a um contador (valor numérico)
- Implementa duas operações básicas: **wait()** e **post()** (ou **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 >>



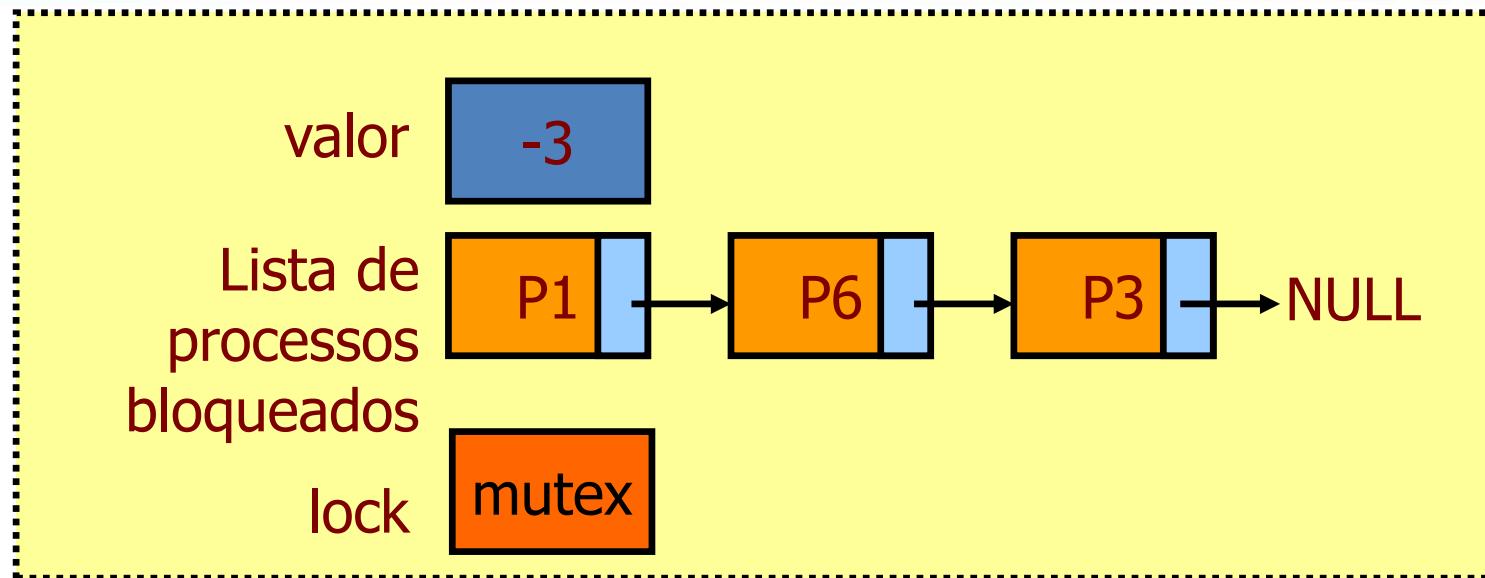
✓ **post()/signal()**

- 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. */  
};
```

Interfaces para semáforos (1)

✓ Semáforos norma POSIX

- 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);
```

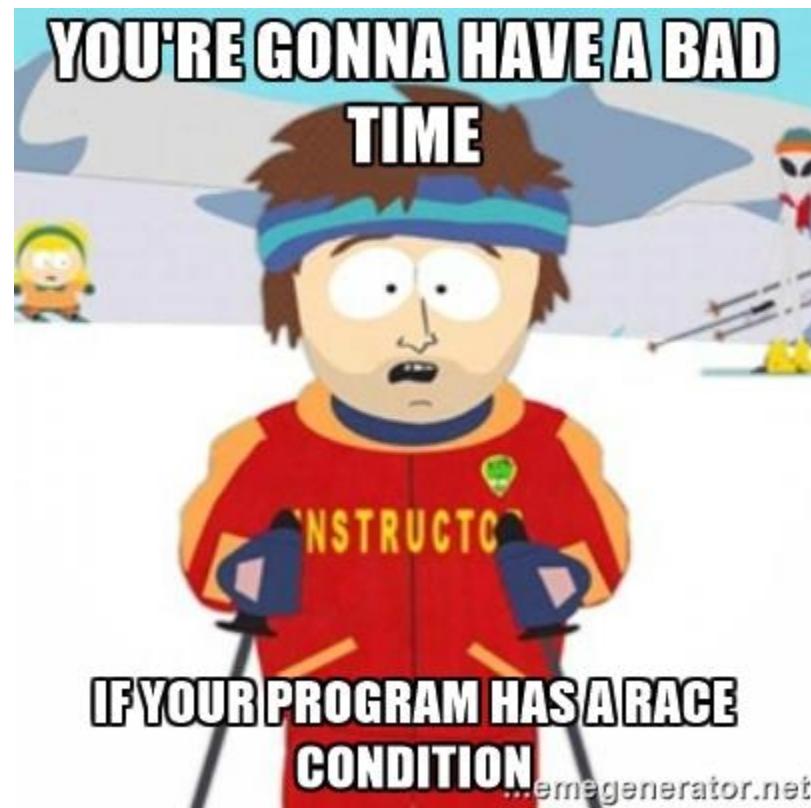
```
int sem_post(sem_t *sem);
```

```
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
 - É uma classe...
 - **System.Threading.Semaphore**



Sistemas críticos

- ✓ São conhecidos vários casos em que situações de race conditions levaram a desfechos trágicos
 - Máquina de radiação Therac-25 (1985-1987)
 - O operador usava uma determinada sequência de teclas para configurar uma sessão de tratamento
 - Ocorria *race condition* na memória
 - Valor afetado correspondia à dose de radiação da sessão
 - Podia ser alterado para mais ou para menos (ocorria também um overflow)
 - Registaram-se vários mortos...
 - Relato e análise ao sucedido: “Medical Devices – The Therac 25”, Nancy Leveson, University of Washington.

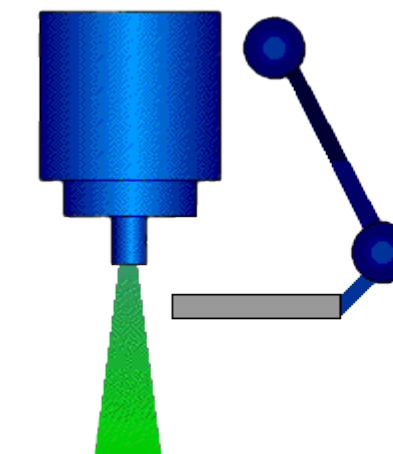
Therac 25 (#1)

- ✓ Therac 25
 - Acelerador linear empregue em radioterapia. O acelerador produz...
 - Fotões com energia de 25MeV
 - Eletrões com vários níveis de energia
- ✓ Therac 25 baseado no therac 6 (6 MeV) e no therac 20 (20 MeV)
 - Contudo, no Therac 25 o software participa na segurança do dispositivo
 - O Therac 20 tinha hardware limitador (“hardlock”)
 - O Therac 25 usa software para essa limitação

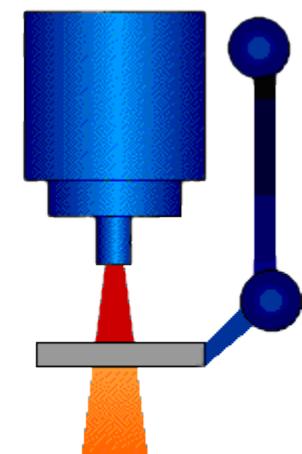
Therac 25 (#2)

✓ Therac 25

- Acelerador linear empregue em radioterapia. O acelerador produz...
 - Fotões com energia de 25MeV
 - Eletrões com vários níveis de energia



Electron Mode



X-Ray Mode

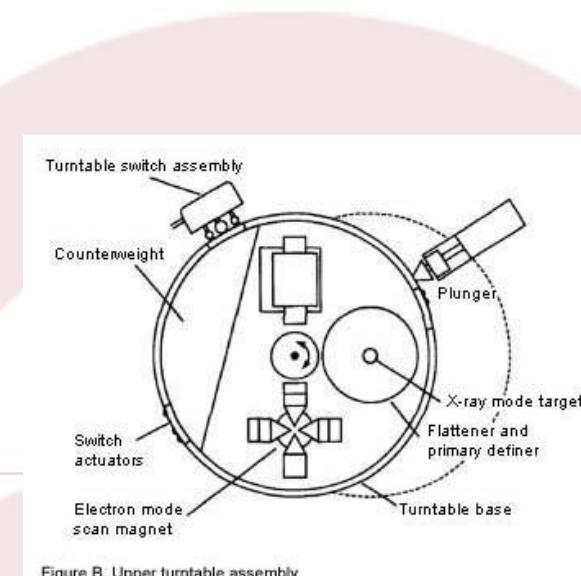
✓ Therac 25 baseado no therac 6 e no therac 20

- Contudo, no Therac 25 o software participa na segurança do dispositivo
 - O Therac 20 tinha hardware limitador da potência
 - O Therac 25 usa software para essa limitação
 - Ambos usam um computador PDP-11 para correr o software



Therac 25 (#3)

- ✓ Nos casos fatais, o Therac-25 apresentou a mensagem “Malfunction 54”
 - Computador do Therac-25's não conseguia determinar se estava em *underdose* ou *overdose*...
 - Foi descoberto padrão de ativação do erro
 - Uso das teclas CIMA/BAIXO para posicionamento do cursor
 - Se fosse selecionado modo raio-x (tecla X), a máquina levava 8 segundos a ativar esse modo
 - Se nesses 8 segundos, o operador trocasse modo para elétrões (tecla E), então o posicionamento do dispositivo de emissão não era o correto
 - O uso do modo X era muito mais frequente, pelo que o operador selecionava X e depois corrigia para E



Therac 25 (#4)

✓ O therac-25 foi programado em assembler

- Tarefa (thread) para tratamento: Treat
- Executa várias subrotinas
 - Uma delas é a Dataent (data entry)
 - Dataent comunica com a tarefa de controlo do teclado (kbd_task) através da variável partilhada dataent_flag
 - As tarefas dataent e kbd_task são concorrentes
 - A tarefa kbd_task deteta que todos os dados foram entrados e ativa a variável dataent_flag
 - A tarefa Dataent deteta a alteração na variável e ativa a próxima fase

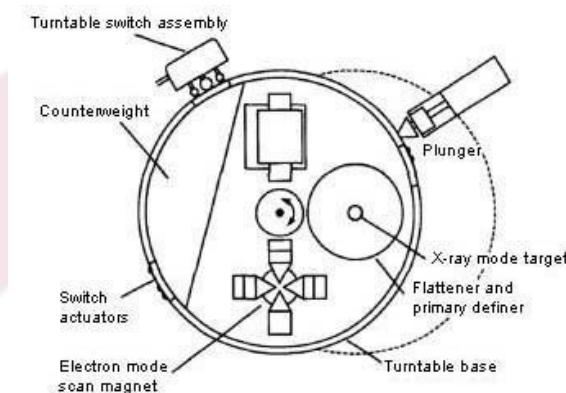


Figure B. Upper turntable assembly

Therac 25 (#5)

✓ Problema ocorre porque durante os 8 segundos para posicionamento

– A tarefa **kbd_task** atualiza a interface com os comandos que o operador possa estar a fazer

- Mas no final dos 8 segundos, a tarefa **Dataent** não lê os valores que entretanto foram inseridos...



PATIENT NAME : TEST	A	1
TREATMENT MODE: FIX	BEAM TYPE: X ENERGY (KeV):	25
UNIT RATE/MINUTE	ACTUAL	PRESCRIBED
MONITOR UNITS	50	200
TIME (MIN)	0.27	1.00
GANTRY ROTATION (DEG)	0.0	0 VERIFIED
COLLIMATOR ROTATION (DEG)	359.2	359 VERIFIED
COLLIMATOR X (CM)	14.2	14.3 VERIFIED
COLLIMATOR Y (CM)	27.2	27.3 VERIFIED
WEDGE NUMBER	1	1 VERIFIED
ACCESSORY NUMBER	0	0 VERIFIED
DATE : 84-OCT-26	SYSTEM: BEAM READY	OP.MODE: TREAT AUTO
TIME : 12:55.8	TREAT : TREAT PAUSE	X-RAY 173777
OPR ID: T25V02-R03	REASON: OPERATOR	COMMAND:

Figure A. Operator interface screen layout.

Pseudo-código...

The Bug

Datent:

```
if mode/energy specified then
begin
    calculate table index
repeat
    fetch parameter
    output parameter
    point to next parameter
until all parameters set
call Magnet
if mode/energy changed then return
end
if data entry is complete then set Tphase to 3
if data entry is not complete then
    if reset command entered then set Tphase to 0
return
```

→ short for "data enter"

indicates
cursor has
been to the
cmd line

Magnet:

```
Set bending magnet flag
repeat
    Set next magnet
    Call Ptime
    if mode/energy has changed, then exit
until all magnets are set
return
```

Ptime:

```
repeat
    if bending magnet flag is set then
        if editing taking place then
            if mode/energy has changed then exit
    until hysteresis delay has expired
    Clear bending magnet flag
return
```

→ bug A: problematic completion check

✓ Mas existia outro erro...

- Uma variável (Class3, 1 byte) com valor zero indicava que se podia prosseguir para o próximo passo
- Com valor diferente de zero, chamava-se rotina de verificação
- A variável Class3 é incrementada para cada verificação
 - Mas todas as 256 vezes Class3 era...0 (*overflow*)
 - Se o operador carregasse na opção SET com a variável Class3 a zero, a dose de radiação era aleatória...

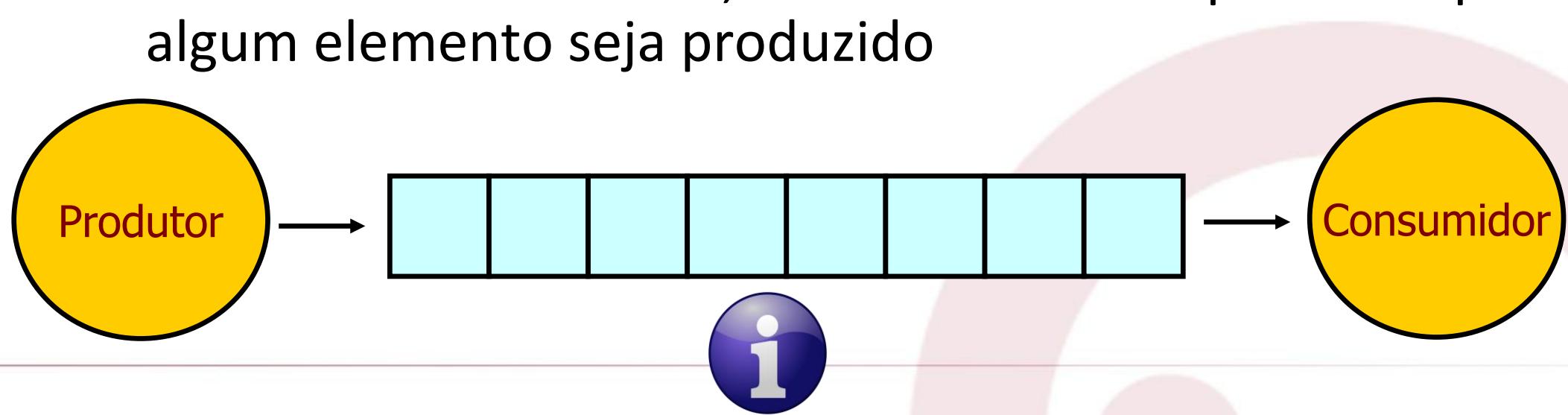
✓ fonte: “Medical devices: The Therac 25”, Nancy Leveson (<http://sunnyday.mit.edu/papers/therac.pdf>)

Problemas clássicos de concorrência

Problema produtor/consumidor

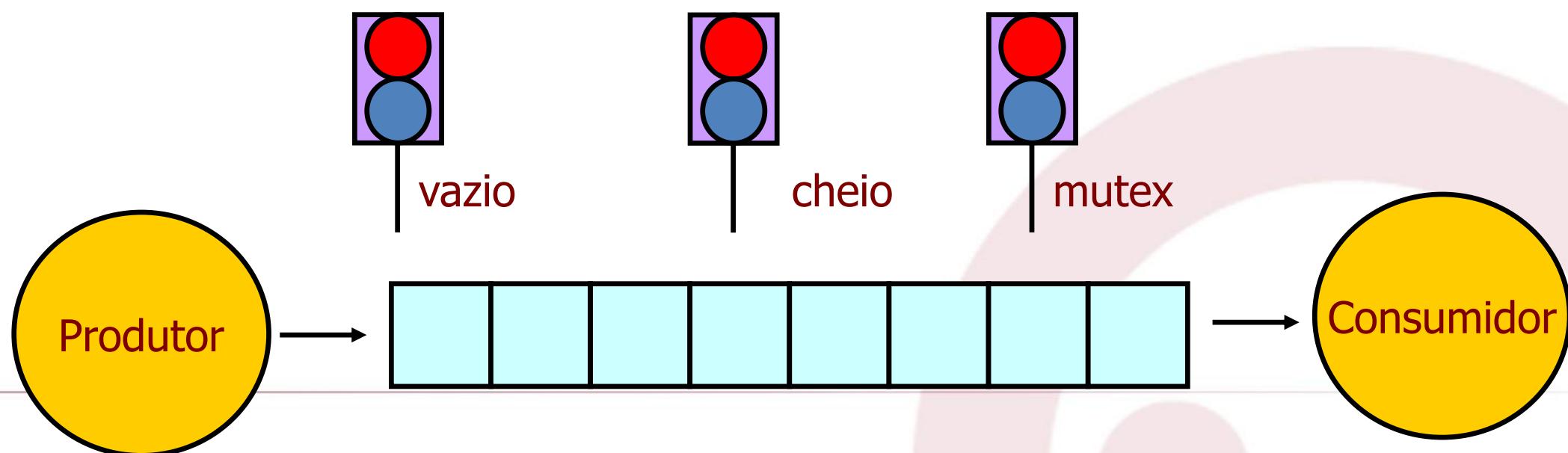
- ✓ Um “produtor” coloca elementos numa zona de memória finita (e.g. vetor)
 - Se o vetor estiver cheio, o produtor bloqueia até que haja espaço livre

- ✓ O consumidor retira (“consome”) elementos
 - Se o vetor estiver vazio, o consumidor bloqueia até que algum elemento seja produzido

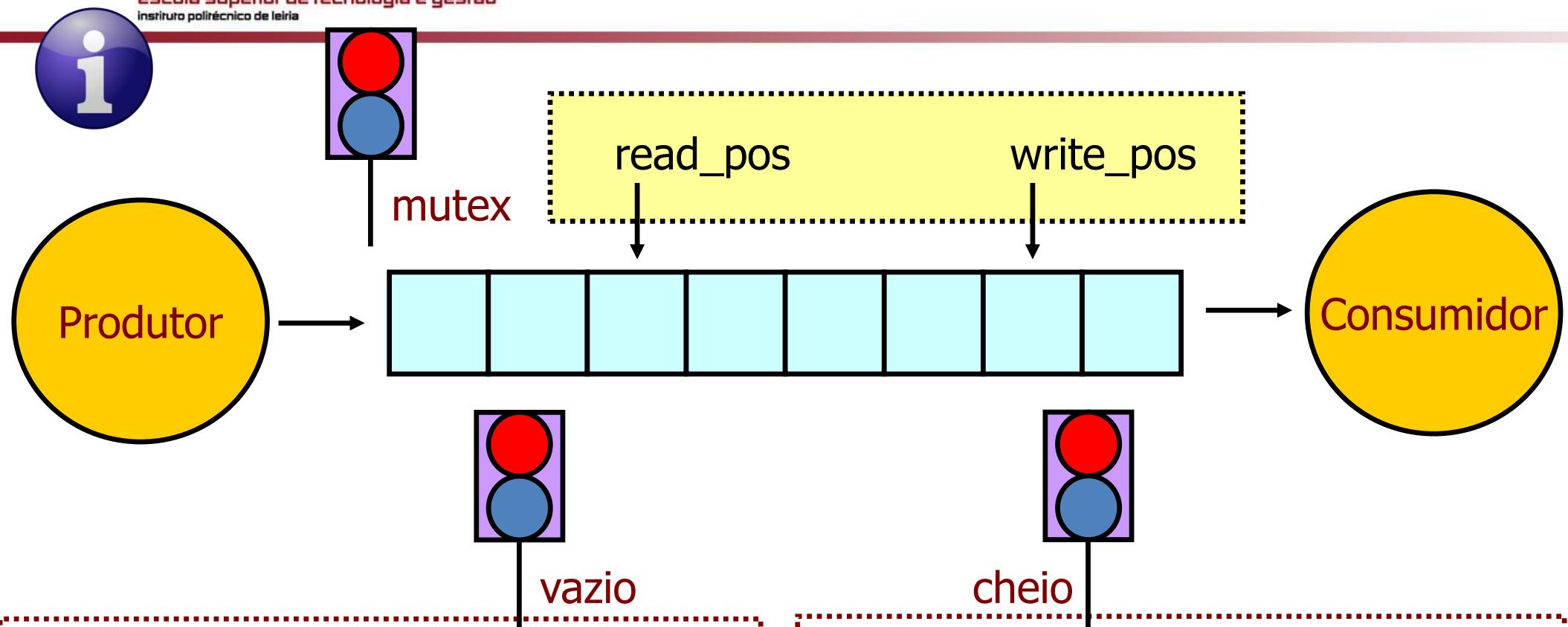


Problema “clássicos” produtor/consumidor

- ✓ 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);  
    }  
}
```

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);

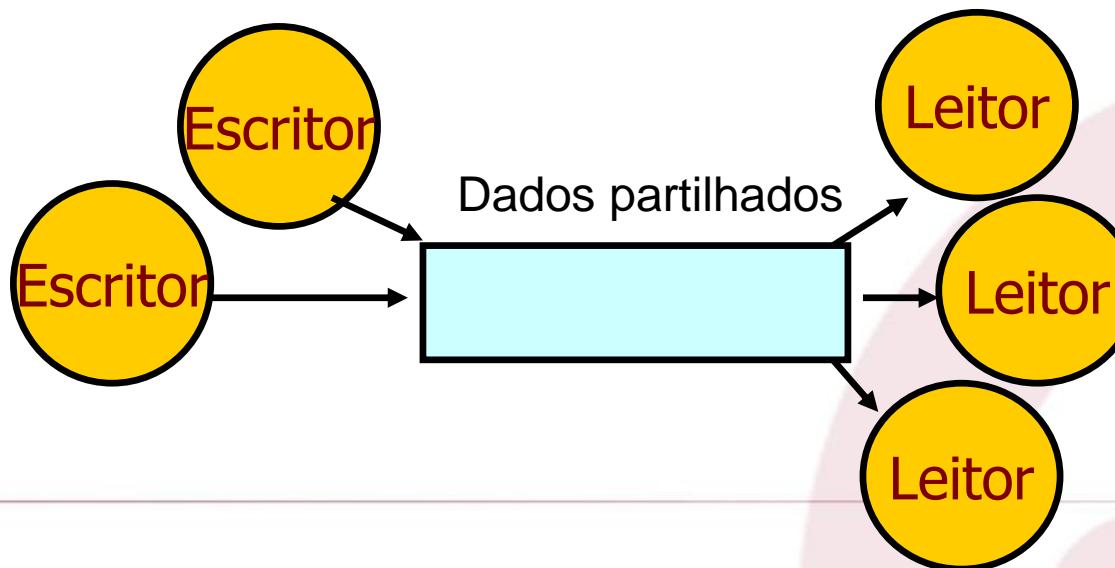
    write_pos = read_pos = 0;

    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 – problema clássico

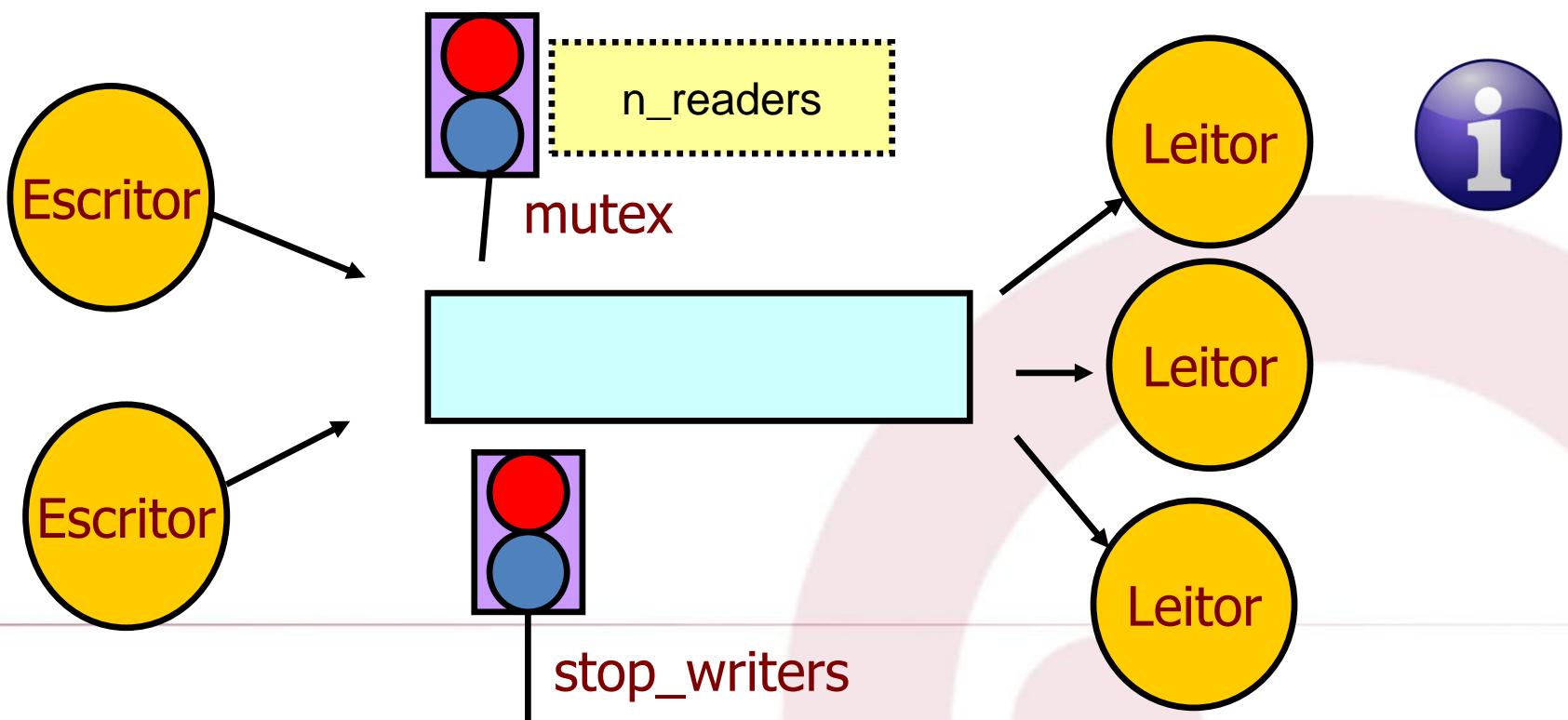
- ✓ Escritores: processos actualizam dados partilhados
- ✓ Leitores: acedem aos dados partilhados
 - Mais do que um processo leitor deve poder aceder simultaneamente aos dados partilhados
- ✓ Em que é que este problema difere do “produtor/consumidor”?
 - Vários escritores/leitores com acesso simultâneo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



Leitores/escritores (2)

✓ São necessários dois semáforos

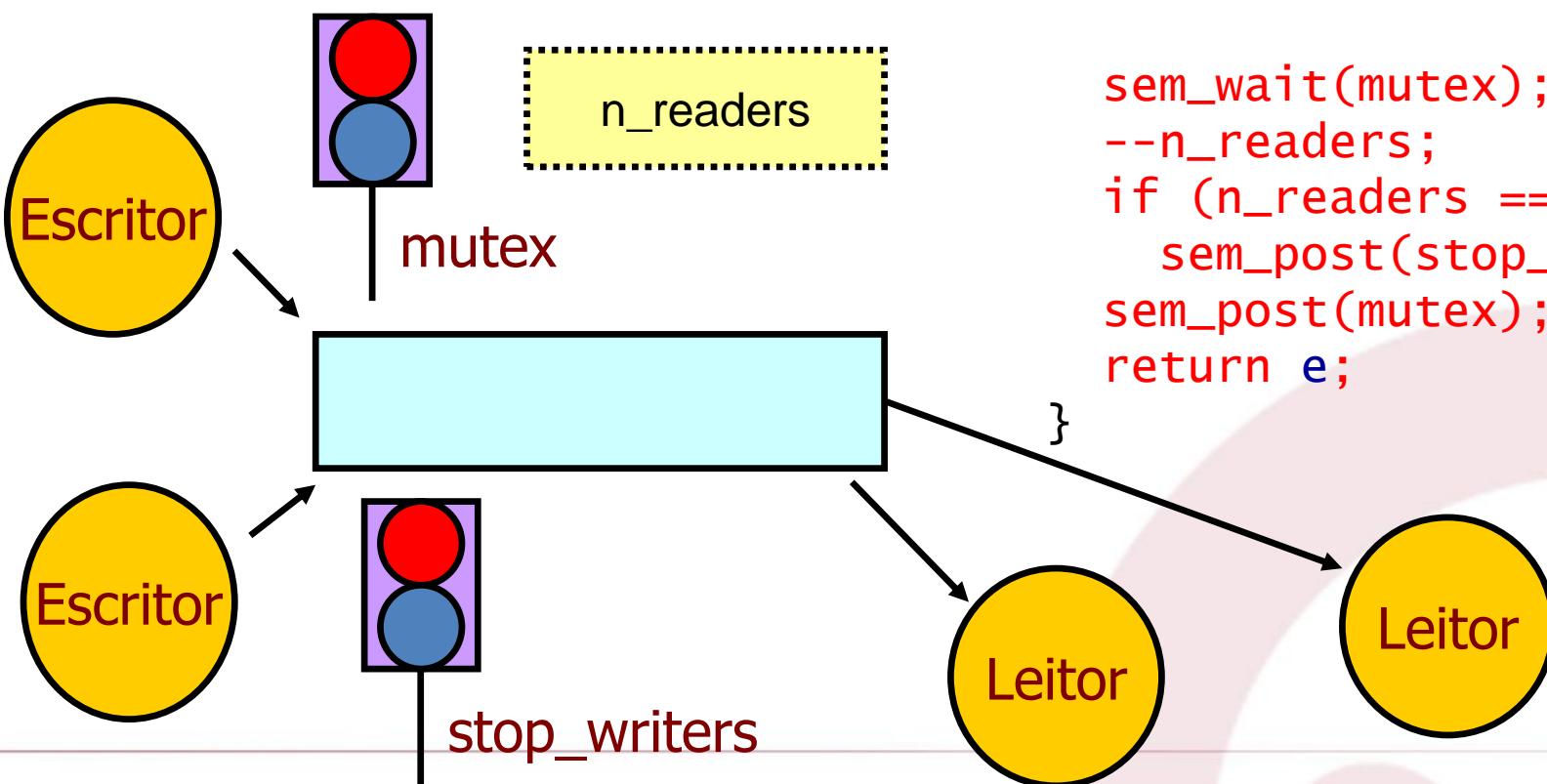
- Um de controlo aos “escritores” (*stop_writers*)
 - Escritores em estado de espera quando já existam leitores
 - Garantir exclusão mútua quando um escritor está a atualizar os dados
- Um para proporcionar exclusão mútua à variável partilhada “contador de leitores” (*n_readers*)



Algoritmo – prioridade aos leitores

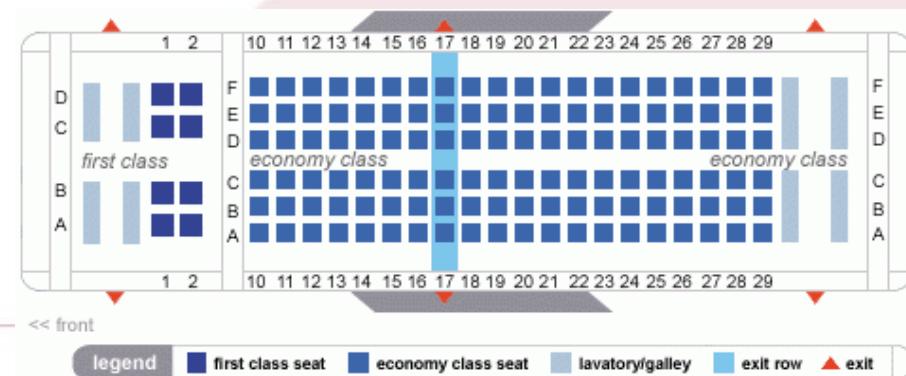
```
/* Escritor */  
write(e) {  
    sem_wait(stop_writers);  
    buffer = e;  
    sem_post(stop_writers);  
}
```

```
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



- ✓ 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
 - t → Sticky bit

- ✓ 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
- ✓ Uso de ficheiros em diretórios temporários
 - Criar o ficheiro de forma atómico
 - 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)
 - Recomenda-se o uso da função `mkstemp`
 - Não usar `mktemp`

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



- ✓ 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)
```



deadlock!

- Processo 2

```
sem_wait(B)
```

```
sem_wait(A)
```

```
// Secção crítica
```

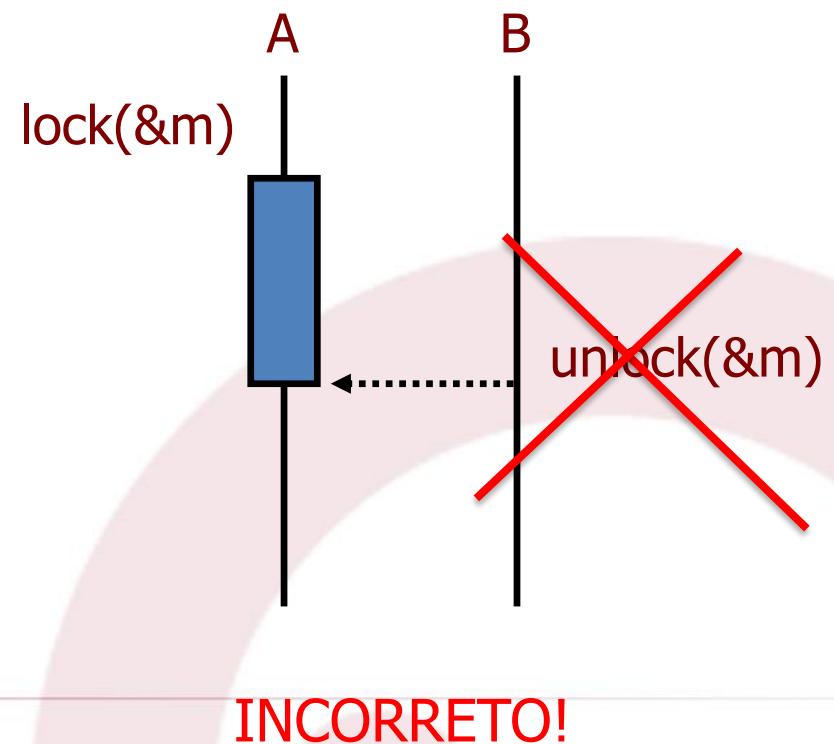
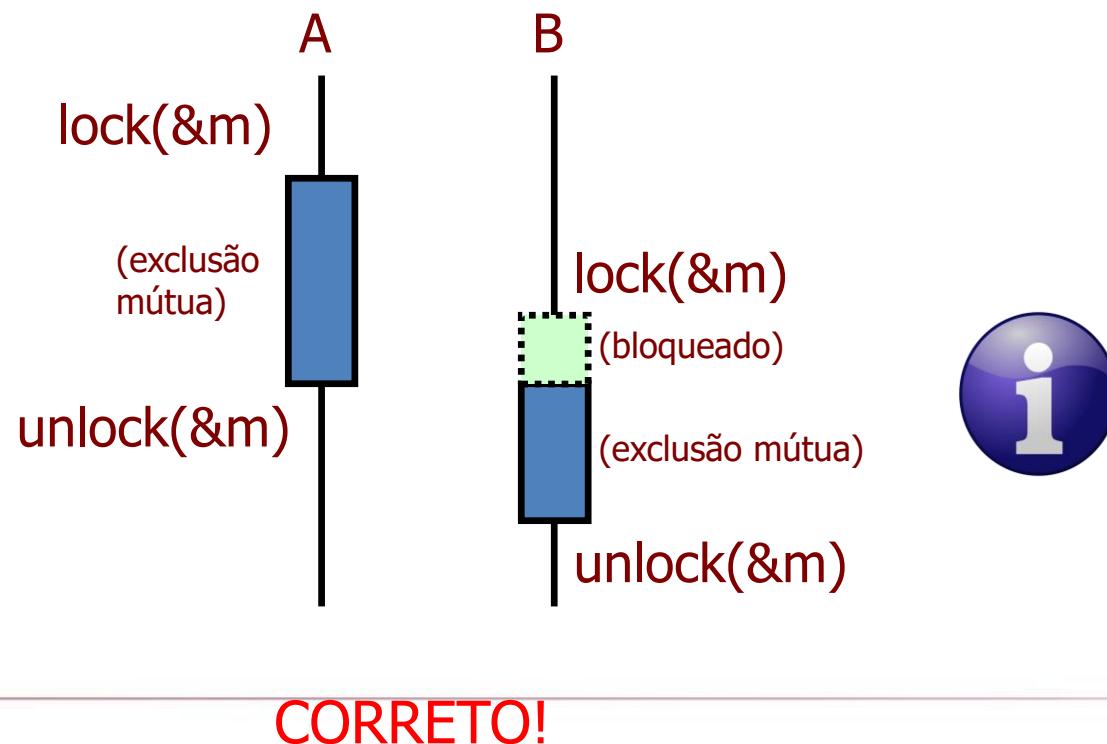
```
sem_post(A)
```

```
sem_post(B)
```

- ✓ 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

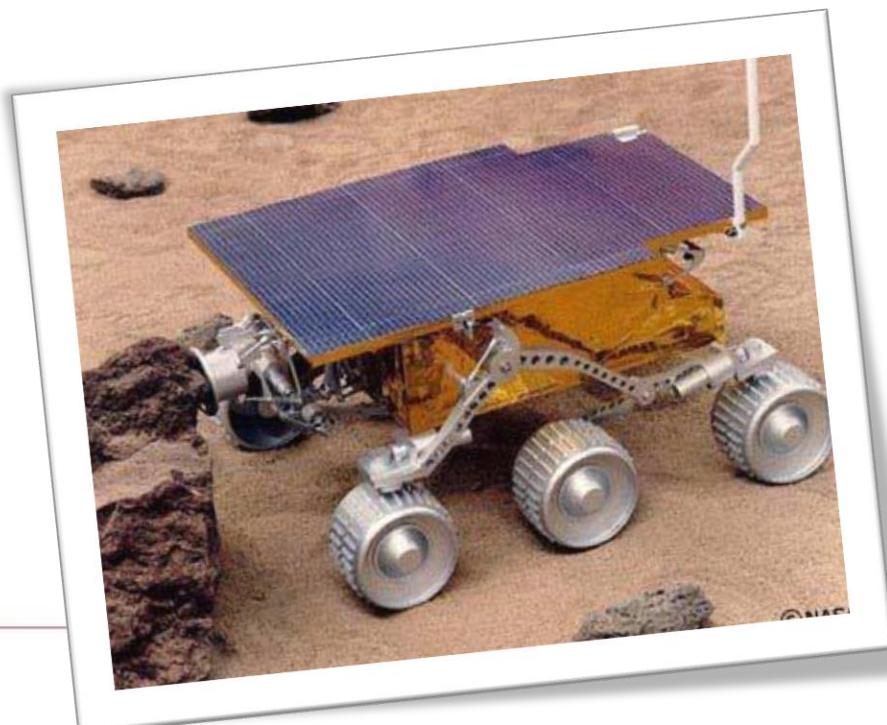
Sincronização – regras básicas...(3)

- ✓ 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!)
- ✓ Assinalar eventos entre threads: usar variáveis de condição!



Sincronização – regras básicas...(4)

- ✓ 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.

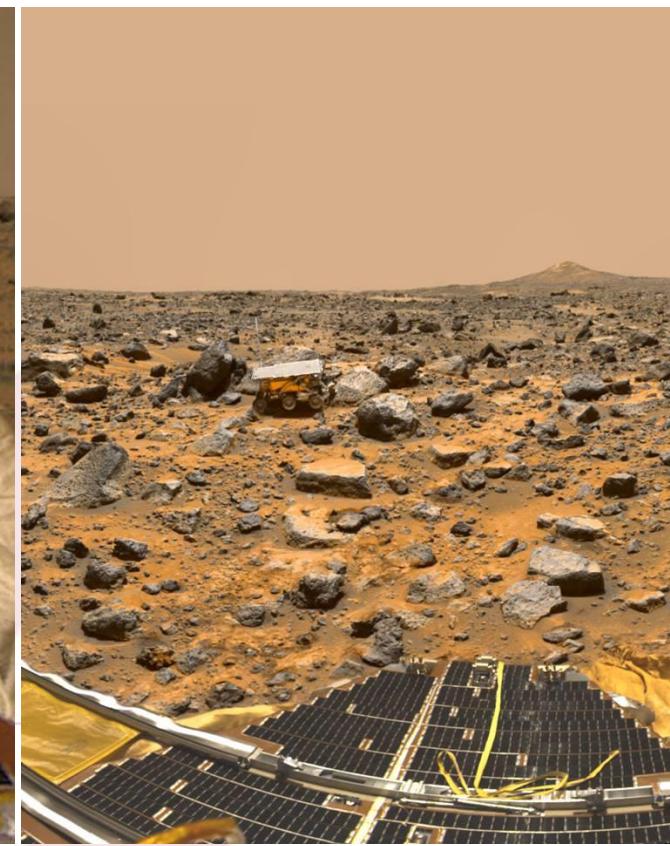
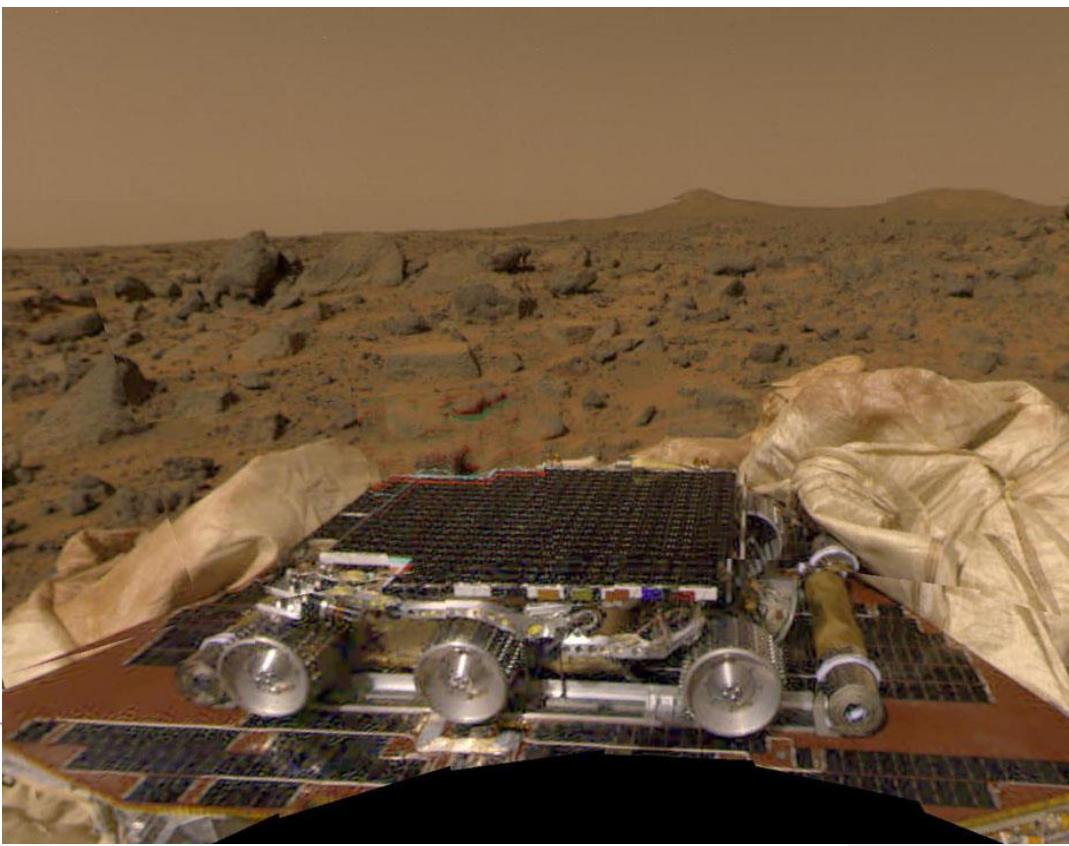
The watch-dog timer was resetting the system because of some unknown reason...”

+info: pesquisar “Sojourner priority inversion problem”

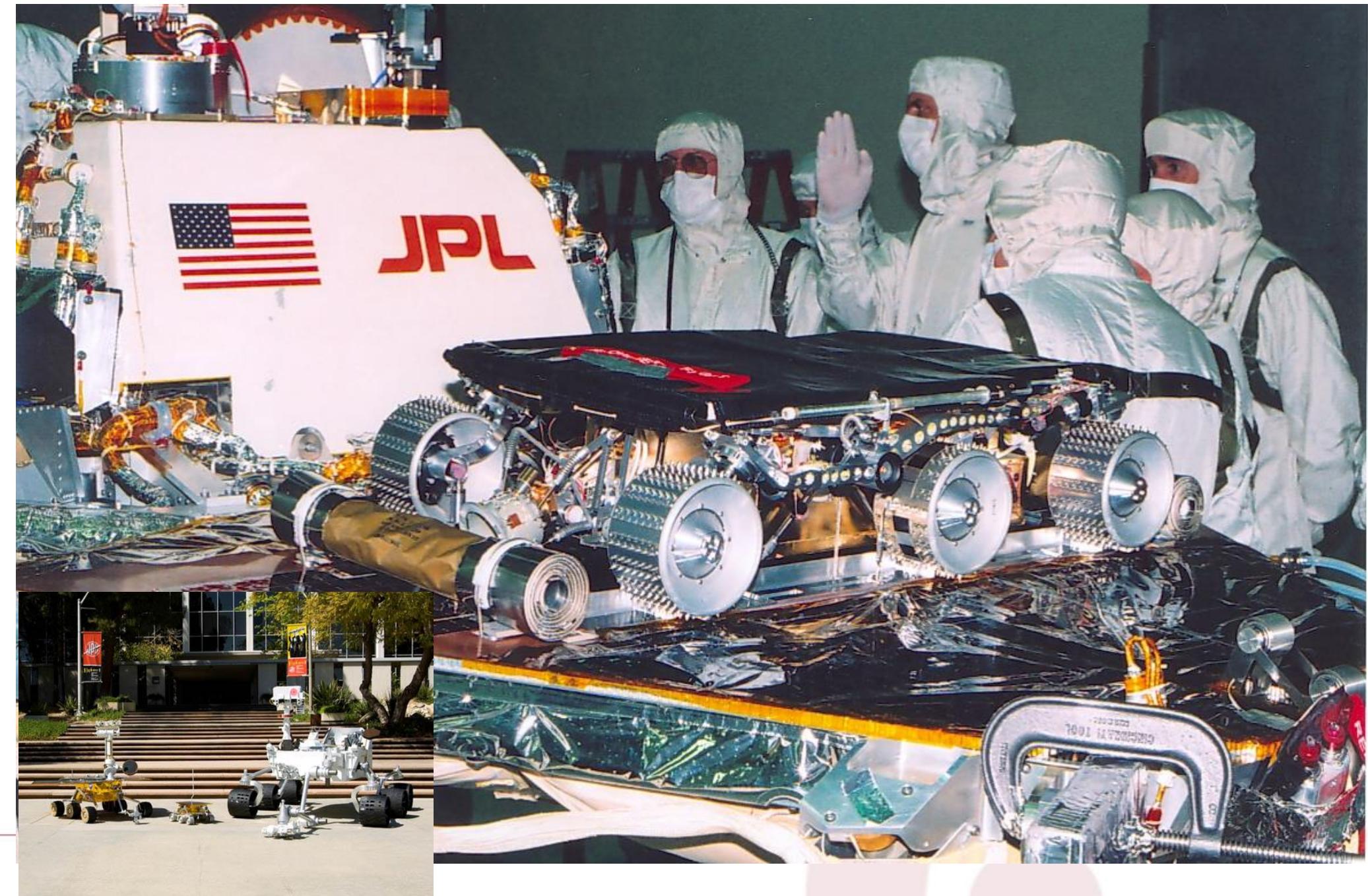
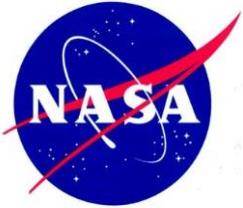
Sojourner (1997)



- ✓ 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.

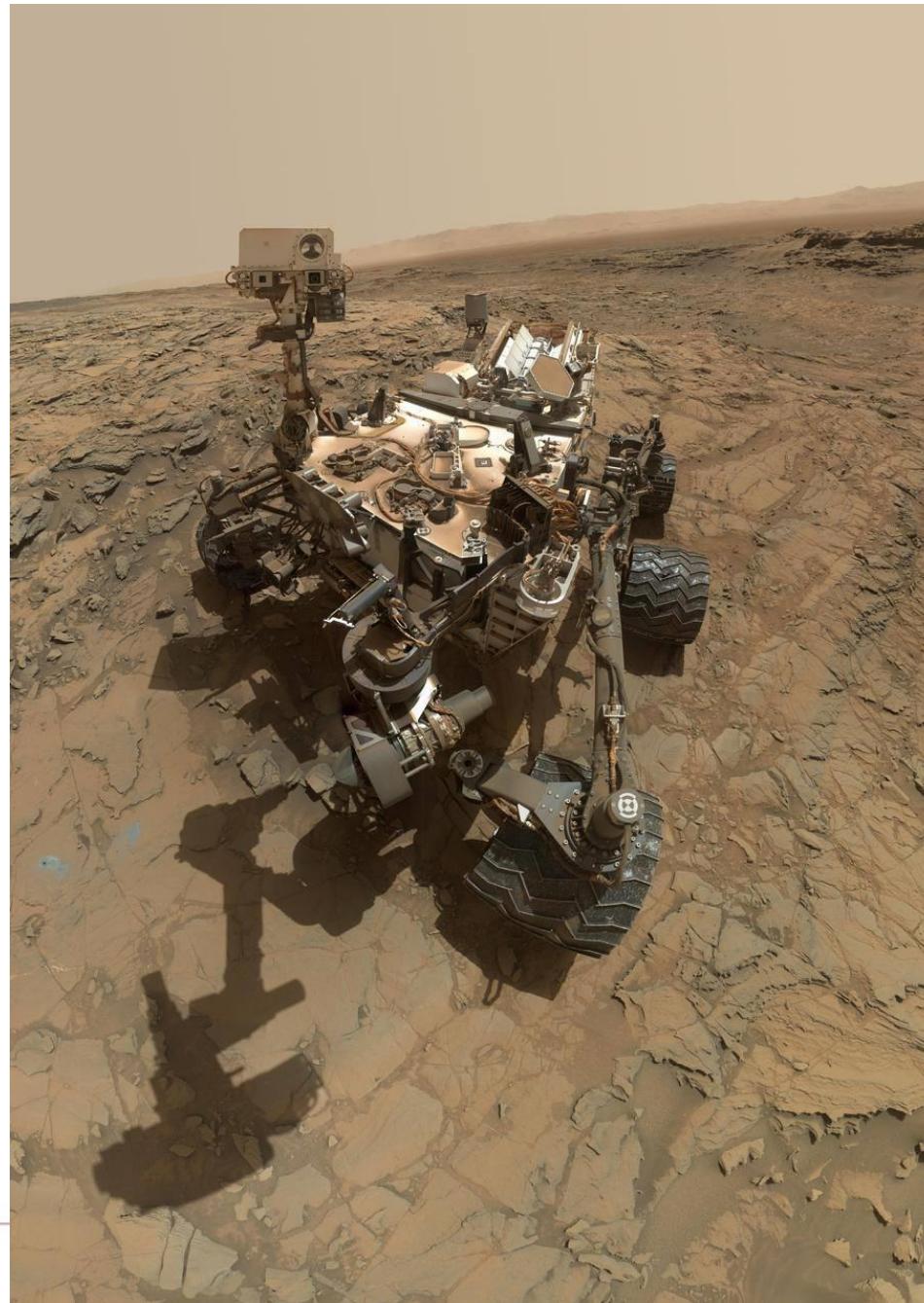


Sojourner (1997)

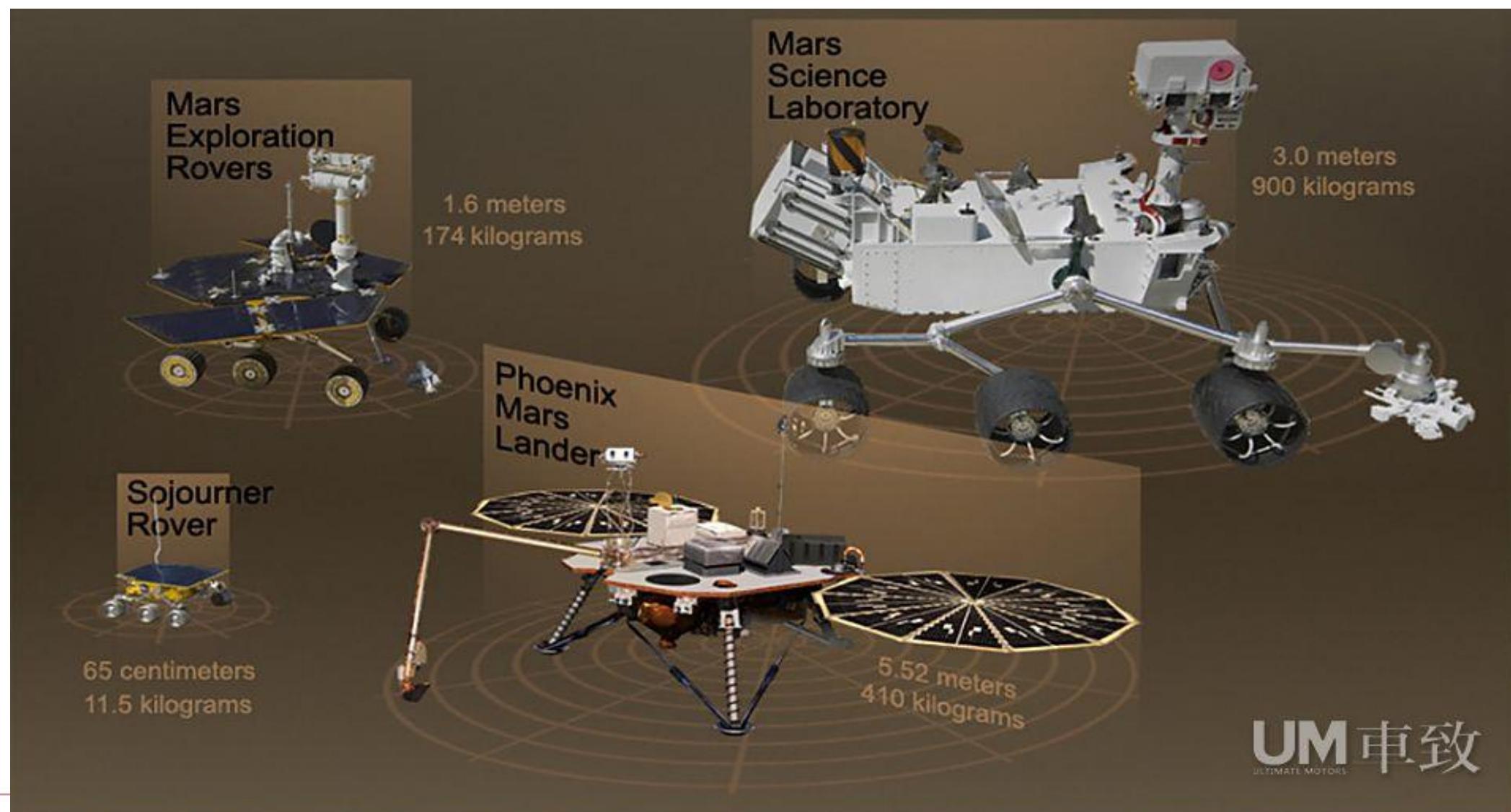


✓ Curiosity

- Em Marte
desde o dia
2012.08.01

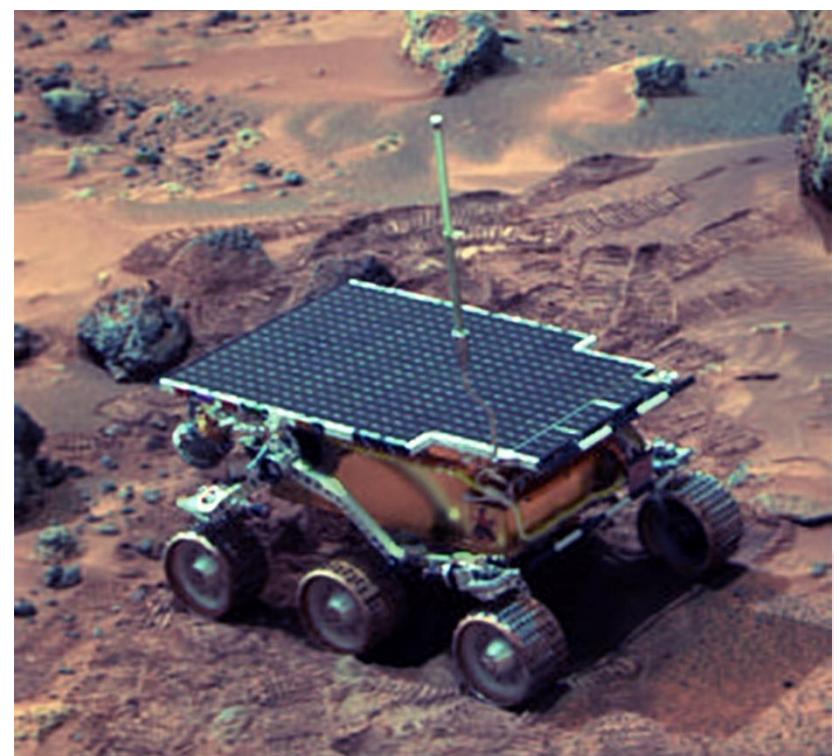
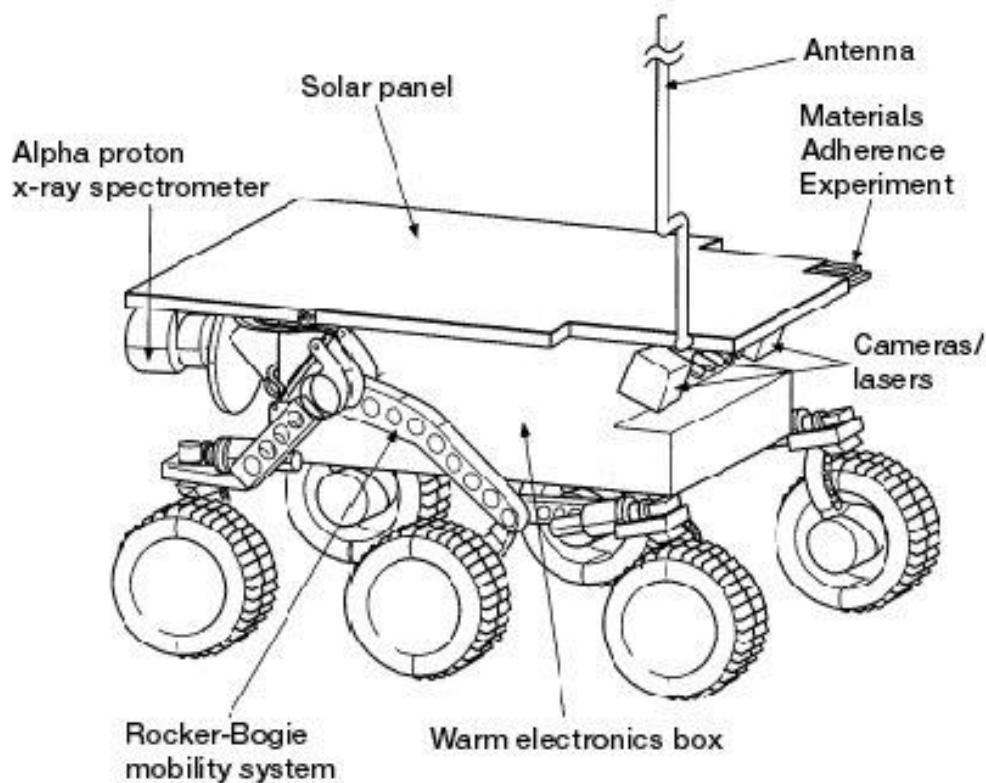


✓ A invasão de Marte...



UM 車致
ULTIMATE MOTORS

✓ Sojourner...em Marte



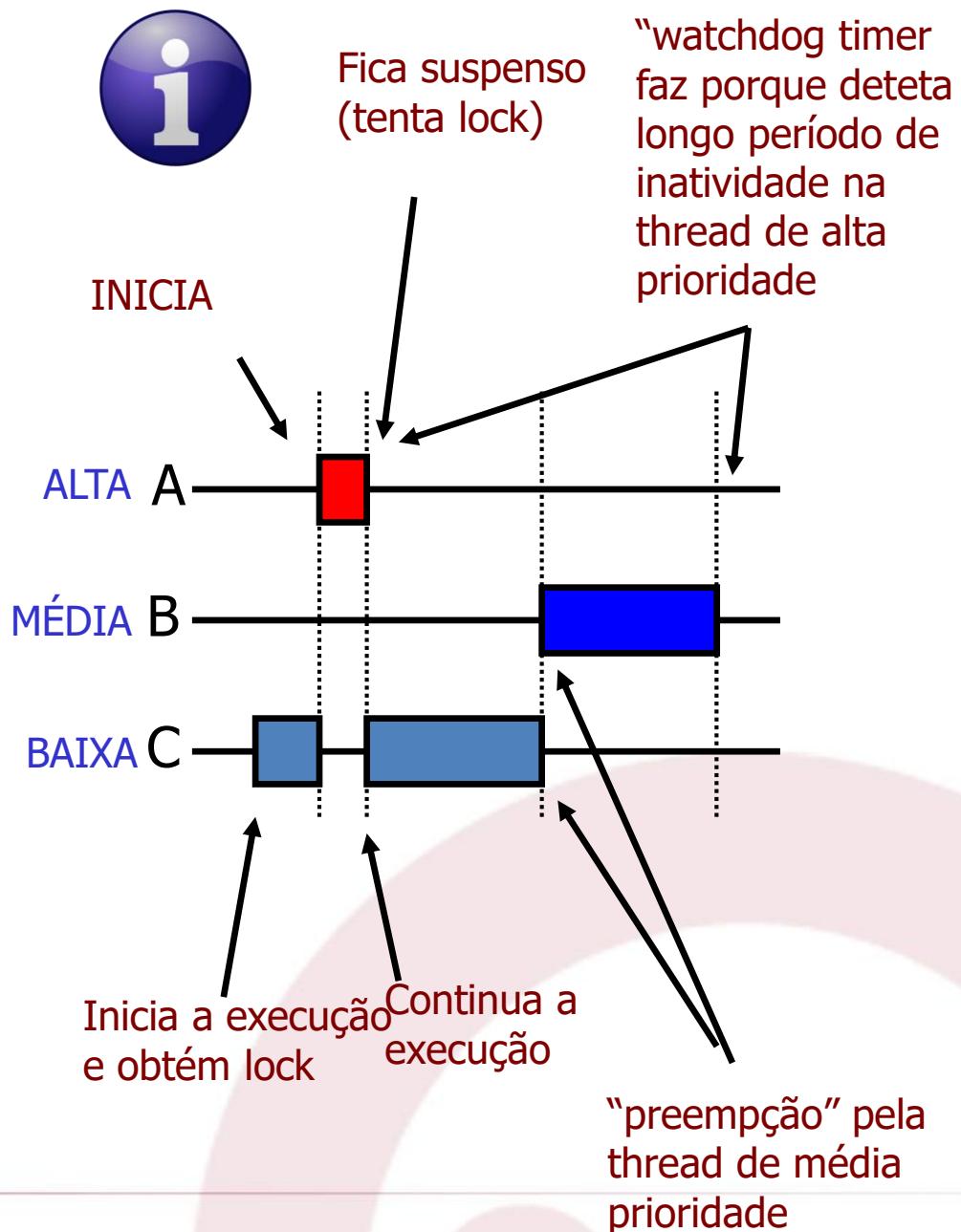
O problema do “Sojourner” (1997)

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”



©NASA



✓ 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

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, 2^a edição, 2013.
- “Operating Systems: three easy pieces” – part 2 – concurrency, Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, 2018.
www.osstep.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)

