:: PROGRAMAÇÃO CONCORRENTE ::
SEMÁFOROS E MONITORES

flaviamsn@ifba.edu.br

COMUNICAÇÃO DE PROCESSOS :: SEMÁFOROS

- Proposto por E. Dijkstra em 1965

- Apesar de ser um mecanismo antigo, ainda é bastante utilizado em programação concorrente.

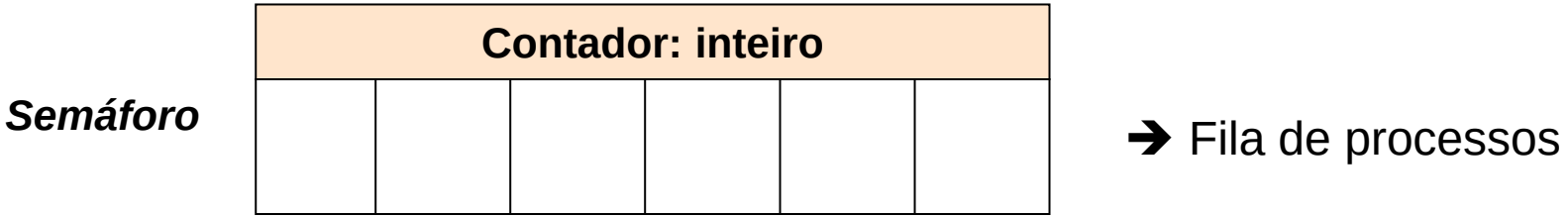
- Na prática, é uma variável que deve ser manipulada de forma ***atômica****
  - A variável possui um contador e uma fila de tarefas;

- Duas primitivas podem ser executadas sobre a variável:
  - *Up()* ➔ *V()*
  - *Down()* ➔ *P()*

COMUNICAÇÃO DE PROCESSOS (-- SEMÁFORO --)

Tipo de dado abstrato:

- Contador: inteiro
- Fila de processos



COMUNICAÇÃO DE PROCESSOS (-- SEMÁFORO --)

Down()

- Decrementa o contador
- solicita acesso à região crítica
  - Livre: processo pode continuar sua execução;
  - Ocupada: processo solicitante é suspenso e adicionado ao final da fila do semáforo;

```
Down(s):
        s.counter--
        if (s.counter <= 0)
        {
           s.enqueue (processo_atual)
           suspend(processo_atual)
        }
```

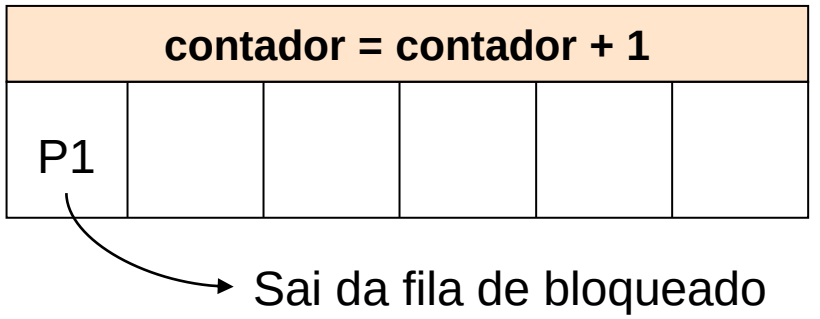
contador = contador - 1					
P1					

COMUNICAÇÃO DE PROCESSOS (-- SEMÁFORO --)

Up ()

- Incrementa o contador
- Liberar a seção crítica
  - Tem processo suspenso: acordar o processo (volta a fila de pronto)
- Chamada é não bloqueante ➔ o processo não precisa ser suspenso para executá-la.

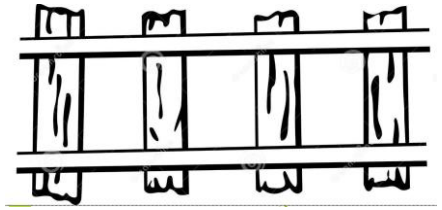
```
Up(s):
        s.counter++
        if (s.counter > 0)
        {
           s.dequeue (processo_atual)
           acorda (processo_atual)
        }
```



SEMÁFOROS :: EXEMPLO ILUSTRATIVO

Recurso:

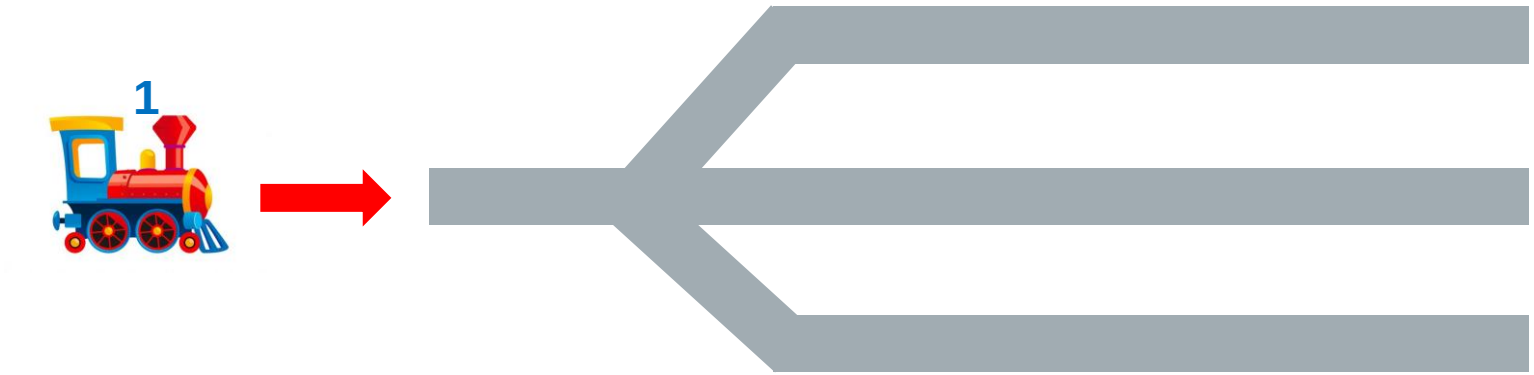
- Vias



Processos:

- trens

SEMÁFORO: EXEMPLO ILUSTRATIVO



Semáforo **s = 3**

Funcionamento do semáforo:

V(s): s = s+1; *(up)*

P(s): **se** s > 0 **então** *(down)*

s = s-1;

SEMÁFORO: EXEMPLO ILUSTRATIVO

Semáforo s = 3
Funcionamento do semáforo:
v: s = s+1;

p: **se** s > 0 **então**
s = s-1;

P(s): **se** s > 0 **então**
s = s-1;

s = 3-1 = 2

SEMÁFORO: EXEMPLO ILUSTRATIVO

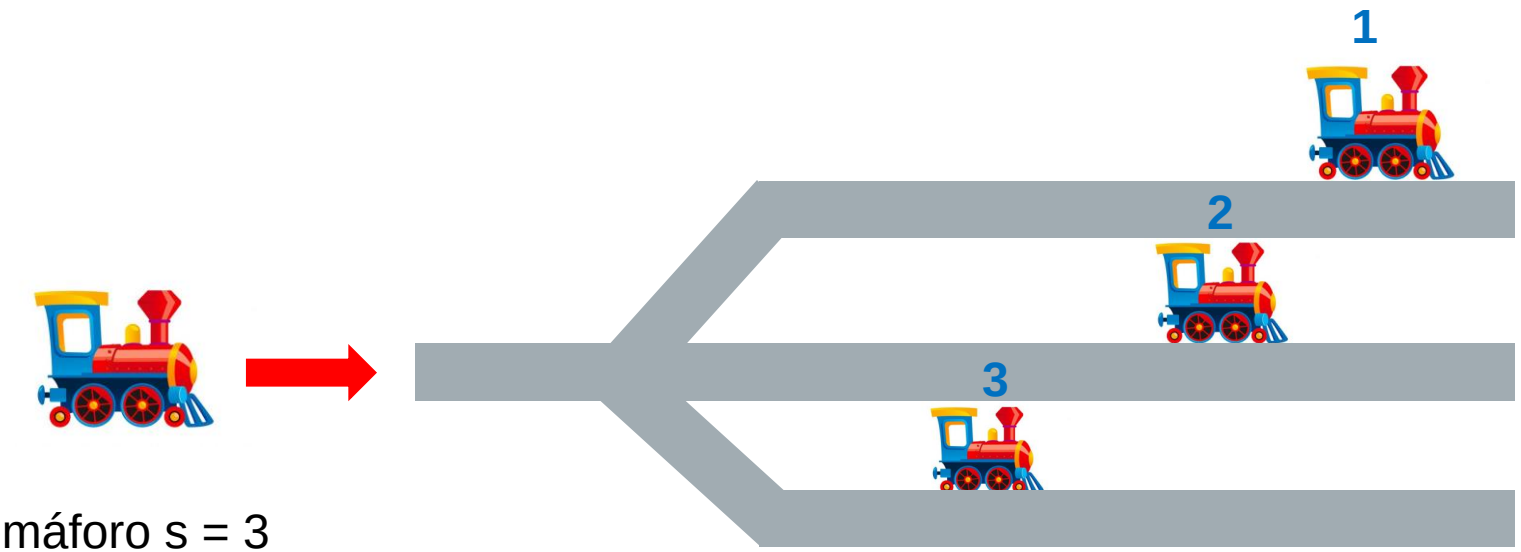
Semáforo s = 3
Funcionamento do semáforo:
V(s): s = s+1;

P(s): **se** s > 0 **então**
s = s-1;

P(s): **se** s > 0 **então**
s = s-1;

s = 2-1 = 1

SEMÁFORO: EXEMPLO ILUSTRATIVO



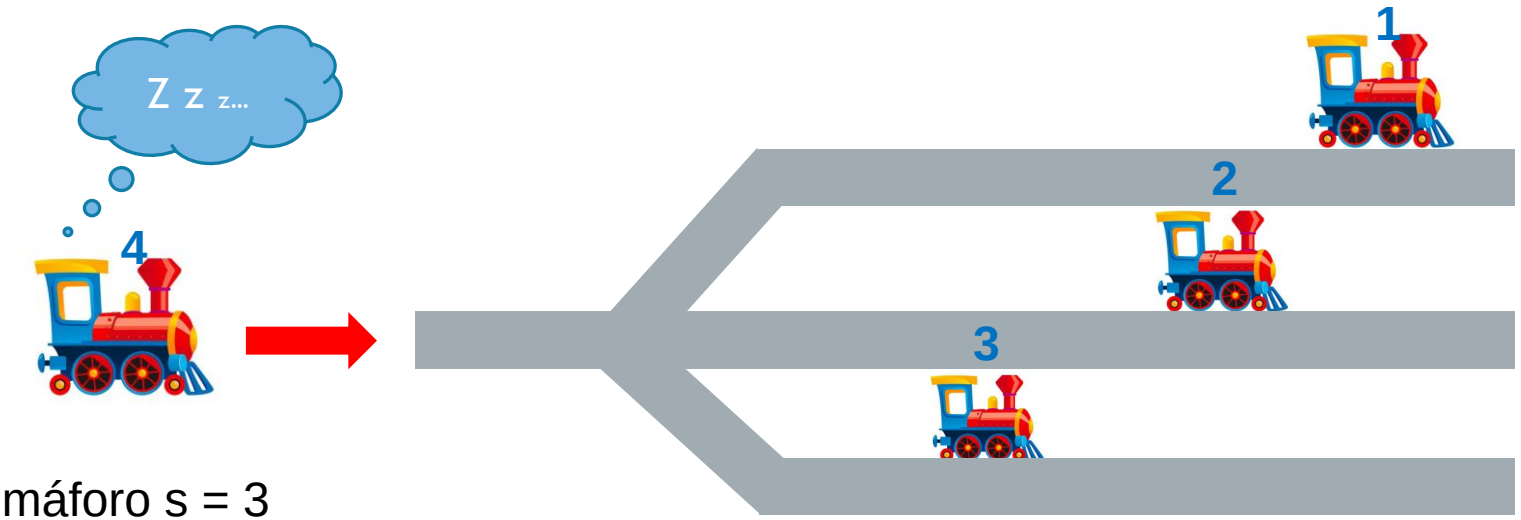
Semáforo s = 3
Funcionamento do semáforo:
V(s): s = s+1;

P(s): **se** s > 0 **então**
s = s-1;

P(s): **se** s > 0 **então**
s = s-1;

s = 1-1 = 0

SEMÁFORO: EXEMPLO ILUSTRATIVO



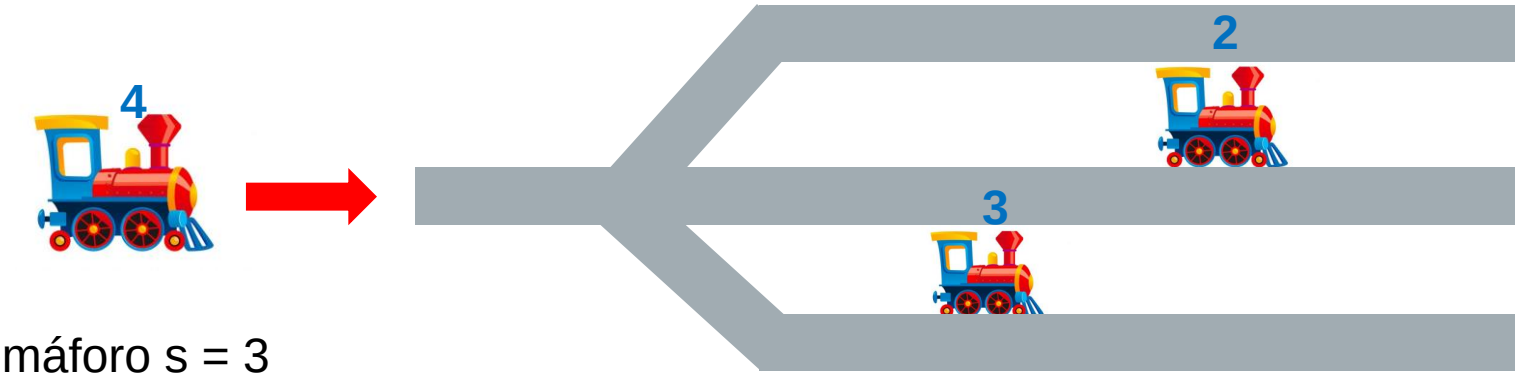
Semáforo s = 3
Funcionamento do semáforo:
V(s): s = s+1;

P(s): **se** s > 0 **então**
s = s-1;

P(s): **se** s > 0 **então**
s = s-1;

s = 1-1 = 0

SEMÁFORO: EXEMPLO ILUSTRATIVO



Semáforo s = 3
Funcionamento do semáforo:
V(s): s = s+1;

P(s): **se** s > 0 **então**
s = s-1;

V(s): s = 1 (trem 1)

P(s): **se** s > 0 **então**
s = s-1;

s = 1-1 = 0

MUTEX :: EXEMPLO ILUSTRATIVO ::

Instante t	Processo	Operação	Executando RC	Bloqueado em s	Valor de s
0					1
1	P1	P(s)	P1		0
2	P1	V(s)			1
3	P2	P(s)	P2		0
4	P3	P(s)	P2	P3	0
5	P4	P(s)	P2	P3,P4	0
6	P2	V(s)	P3	P4	0
7			P3	P4	0
8	P3		P4		0
9	P4	V(s)			1

FIGURA 2.28 O problema do produtor-consumidor usando semáforos.

```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE e a constante 1 */
          item = produce_item( );                /* gera algo para por no buffer */
          down(&empty);                          /* decresce o contador empty */
          down(&mutex);                          /* entra na regiao critica */
          insert_item(item);                     /* poe novo item no buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&full);                             /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* laco infinito */
          down(&full);                           /* decresce o contador full */
          down(&mutex);                          /* entra na regiao critica */
          item = remove_item( );                 /* pega item do buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&empty);                            /* incrementa o contador de lugares vazios */
          consume_item(item);                    /* faz algo com o item */
     }
}
```

RC

RC

Mutual exclusion

MONITORES

Uma abstração de alto nível que provê um mecanismo eficiente para a sincronização de processos;

Apresenta uma série de operações definidas pelo programador;

O acesso a região crítica é feito via monitor, que provê exclusão mútua.

MONITORES :: ESTRUTURA

Contém um conjunto de declaração de variáveis e métodos que operam sobre as variáveis compartilhadas;

```
monitor monitor_name{
//variaveis compartilhadas
   procedure P1(...) {
   ....
   }

   procedure P2(...) {
   ....
   }

   initialization_code(...) {
   ....
   }
}
```

- Procedimentos dos monitores só podem acessar as varáveis de dentro dos monitores
- Variáveis compartilhadas só podem ser acessadas através destes procedimentos
- A construção do monitor garante que apenas um processo por vez pode estar ativo no monitor

MONITORES :: ESTURUTURA

A condição de construção do monitor usa variáveis do tipo "*condition variables*";

Estas variáveis usam dois métodos:

- ***wait()*:** responsável por suspender a execução de um processo;
- ***signal()*:** *responsável por* liberar processos, anteriormente suspensos

Fila de entrada do monitor