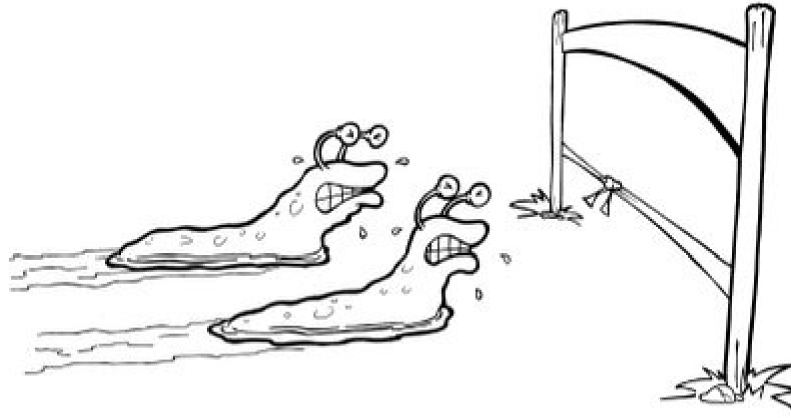
Comunicação entre processos
(-- *Race condition* --)



- Dois processos podem tentar ler ou escrever dados num espaço compartilhado, e o resultado final depende de quem está executando naquele momento.

Comunicação entre processos (-- *Race condition* e região crítica --)

- O que causa condição de corrida?
  - QUALQUER TIPO DE COMPARTILHAMENTO!!

- O trecho de código em que a memória compartilhada é acessada é chamado de <u>região crítica.</u>

P_1:

```
x := x + 1
y := 5 + 2
z := y + t
```

P_2:

```
x := x * 2
a := 2 * 5
c :=  a - 7
```

Considerando x = 2

$P_1 \rightarrow P_2$: x = 6

$P_2 \rightarrow P_1$: x = 5

Comunicação entre processos (-- *Race condition* e região crítica --)

- Como evitar condições de corrida?
  - Sincronizando os processos

ou seja

  - Proibindo que mais de um processo possa ler ou escrever numa área compartilhada ao mesmo tempo.

Comunicação entre processos (-- Exclusão Mútua --)

- Definição:
  – Mecanismo que garante que cada processo que usa uma área compartilhada terá acesso exclusivo a mesma.

Qual é o problema da exclusão mútua??

Para pensar...

- Pense no problema do PRODUTOR *vs*. CONSUMIDOR.

- O que acontece se quando o produtor estiver armazenando um item, o consumidor não puder consumir nada?

Comunicação de Processos
(-- Exclusão mútua e região crítica --)

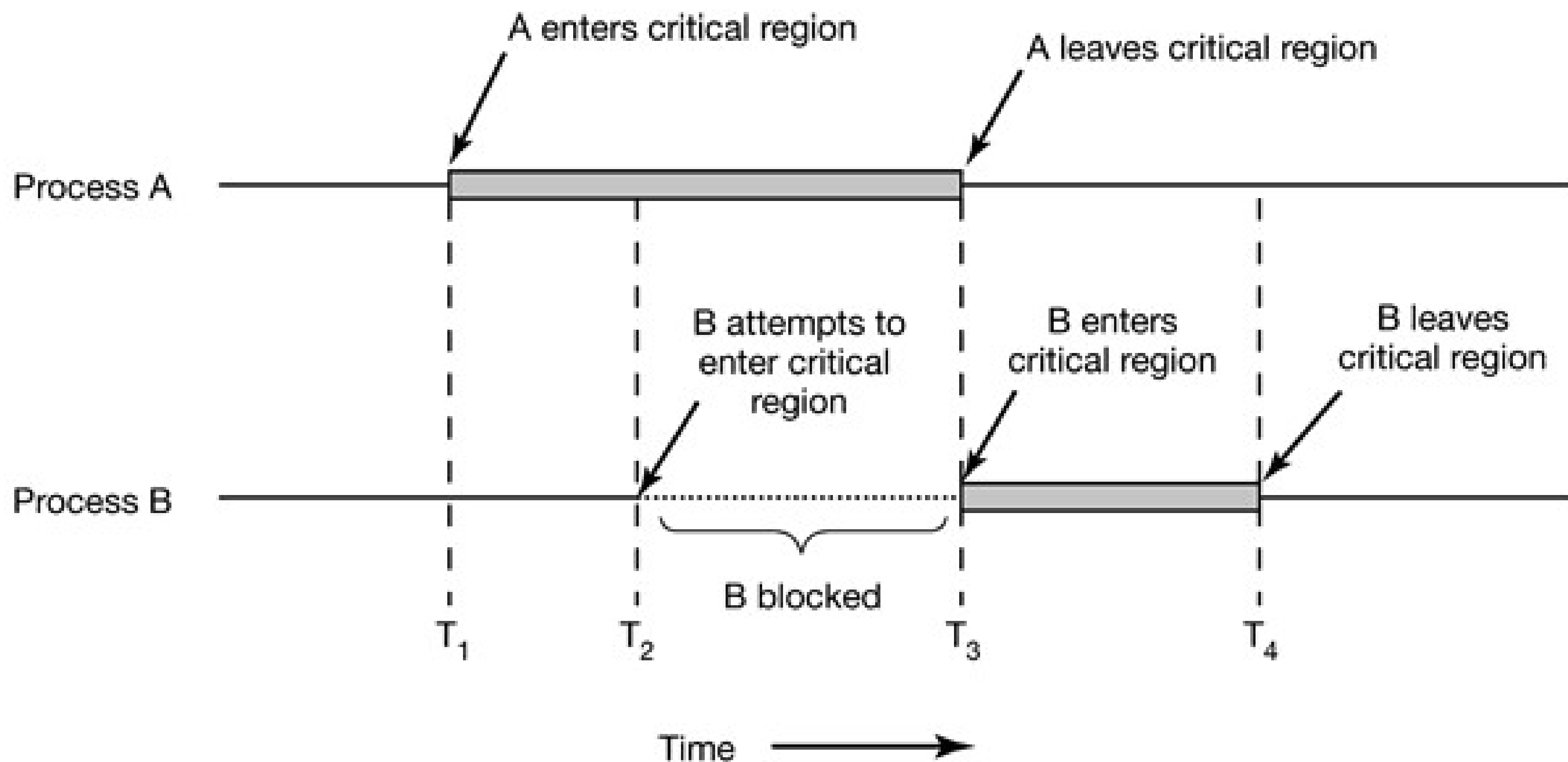
- Dois processos não podem estar simultaneamente em suas regiões críticas

- Nada pode ser assumido com relação a velocidade dos processos ou quantidade de processadores disponível

- Nenhum processo fora de sua região crítica pode bloquear um processo que esteja na região crítica

- Nenhum processo deve esperar indefinidamente para entrar na região crítica.

Comunicação de Processos
(-- Exclusão mútua e região crítica --)



Tanenbaum, Capítulo 2

Comunicação de Processos
(-- Como implementar exclusão mútua --)

- Espera ocupada
- *Sleep* and *wakeup*
- Semáforos
- Mutex
- Monitores

Comunicação de Processos
(-- Exclusão mútua + espera ocupada --)

- Premissa da espera ocupada:
  - Enquanto um processo executa na região crítica, o outro apenas espera.

- Formas de implementar:
  - Interrupção:
    - Problema: não é ideal que processos tenham controle sobre as interrupções

Comunicação de Processos
(-- Exclusão mútua + espera ocupada --)

- Formas de implementar:
  – Alternância Obrigatória

```
while (TRUE) {
    while (turn != 0)    /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1);  /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

Comunicação de Processos
(-- Sleep e Wakeup --)

- Primitivas (chamadas de sistemas)
- *sleep()*
  - Bloqueia um processo enquanto aguarda um recurso
- *wakeup()*
  - Ativa o processo quando o recurso foi liberado

Comunicação de Processos
(-- *Sleep* e *Wakeup* --)

```
#define N 100           /* number of slots in the buffer */
int count = 0;          /* number of items in the buffer */

void producer (void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

Comunicação de Processos
(-- Semáforo --)

- Proposto por E. Dijkstra em 1965

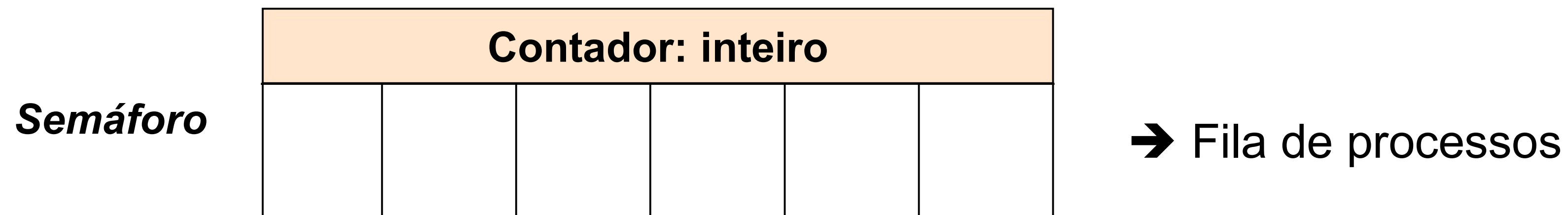
- Apesar de ser um mecanismo antigo, ainda é bastante utilizado em programação concorrente.

- Na prática, é uma <u>variável</u> 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



➔ 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;

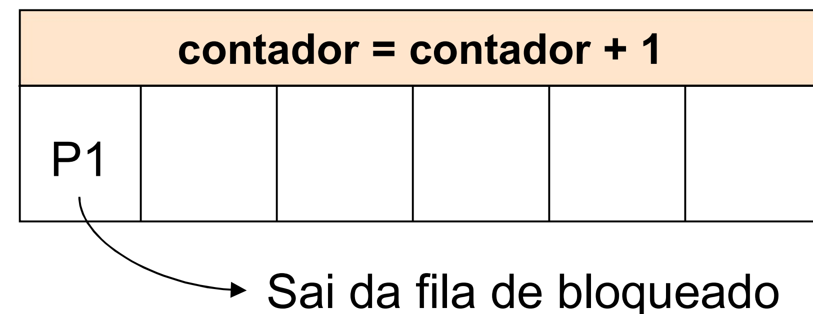
```
Down(s):
    s.counter--
    if (s.counter < 0)
    {
      s.enqueue (processo_atual)
      suspend(processo_atual)
    }
```

contador = contador - 1					
P1					

Comunicação de Processos
(-- Semáforo --)

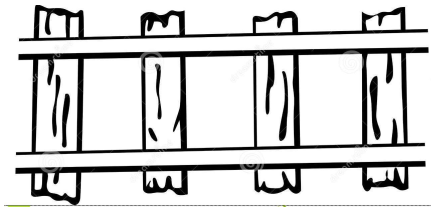
- *Up* ()
  - Incrementa o contador
  - Liberar a seção crítica
    - Tem processo suspenso: acordar o processo (volta a fila de pronto)
  - Chamada é não bloqueante ➔ o processo não precisa ser suspenso para executá-la.

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



Controle de acesso a recurso compartilhado

- 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

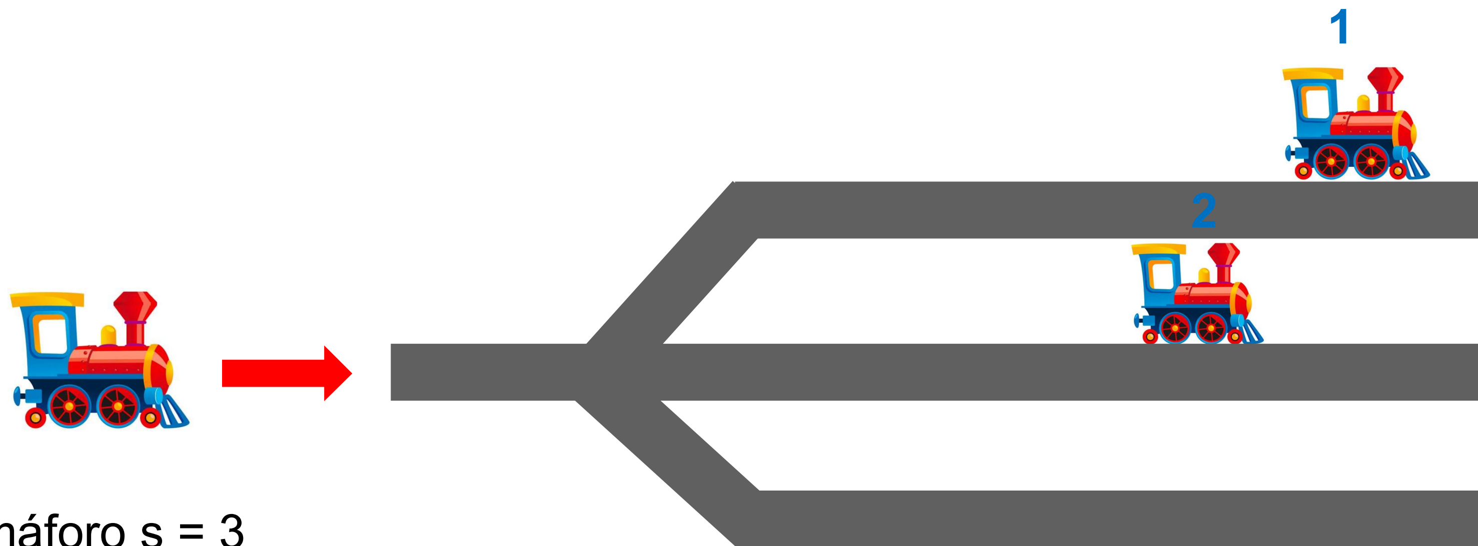
Semáforo s = 3
Funcionamento do semáforo:
v: s = s+1;

p: **se** s > 0 **então**
    s = s-1;

P(s): **se** s > 0 **então**
    s = s-1;

s = 3-1 = 2

Semáforo: Exemplo ilustrativo



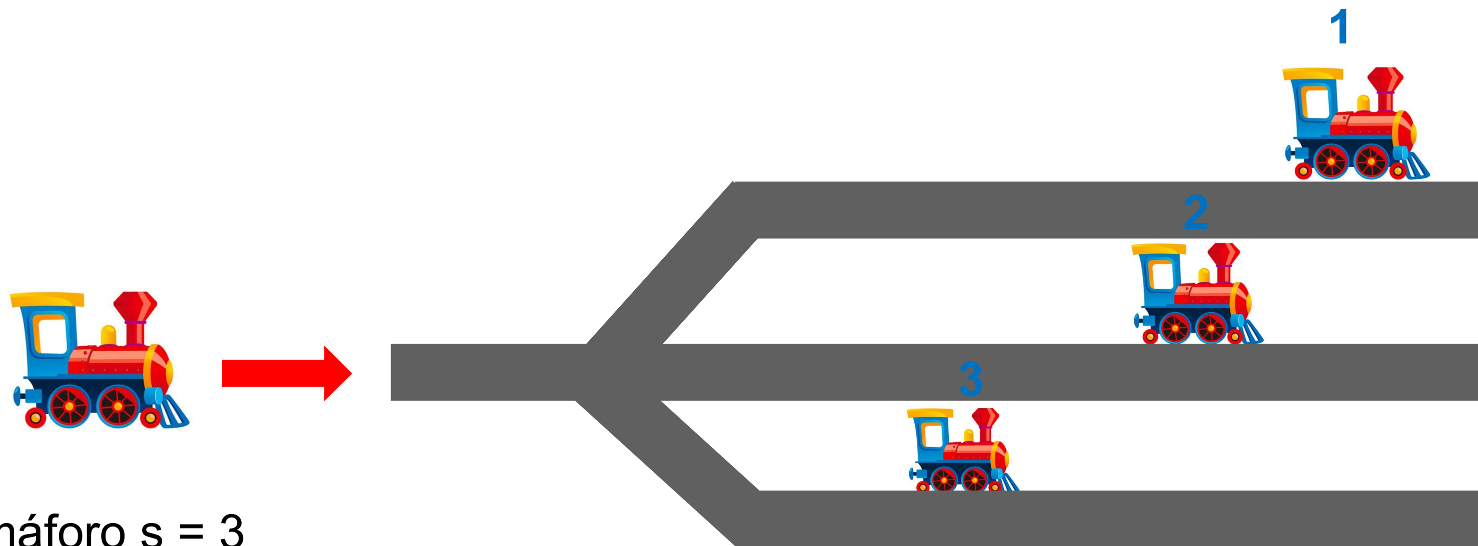
Semáforo s = 3
Funcionamento do semáforo:
V(s): s = s+1;

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



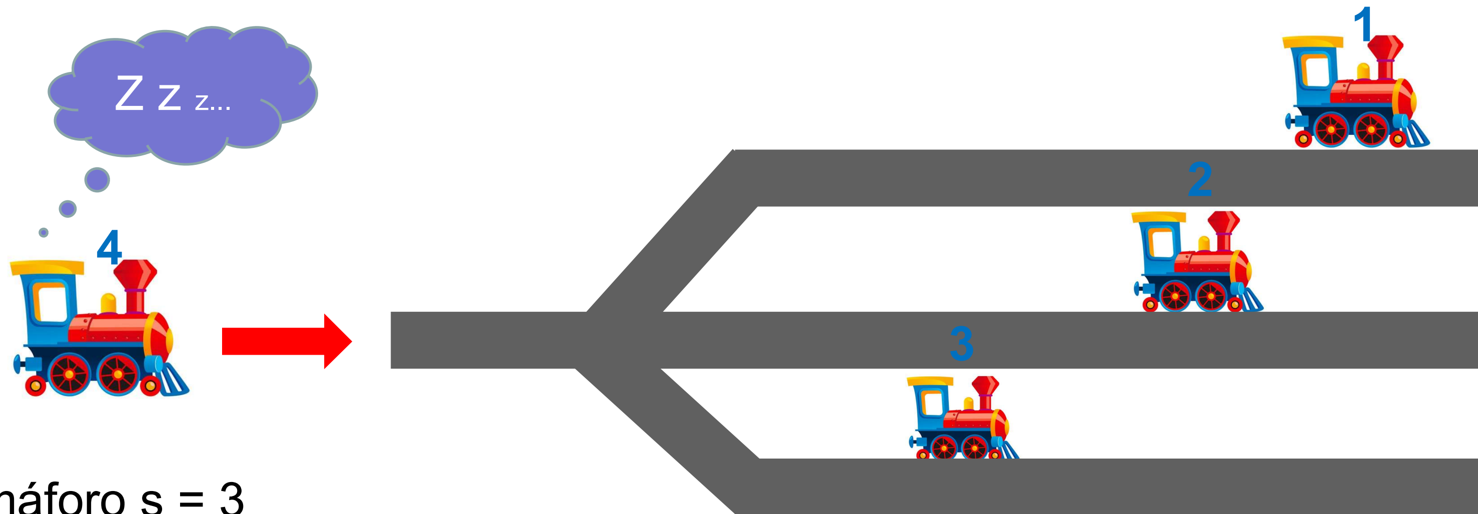
Semáforo s = 3
Funcionamento do semáforo:
V(s): s = s+1;

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:

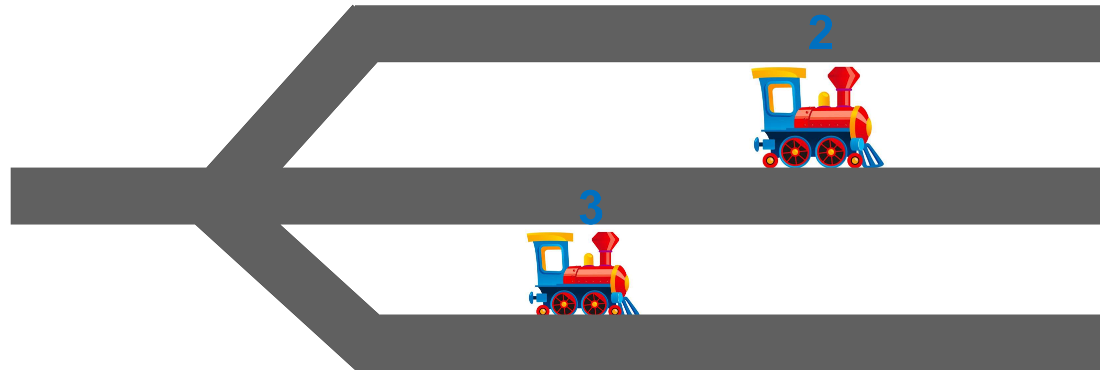
V(s): s = s+1;

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)

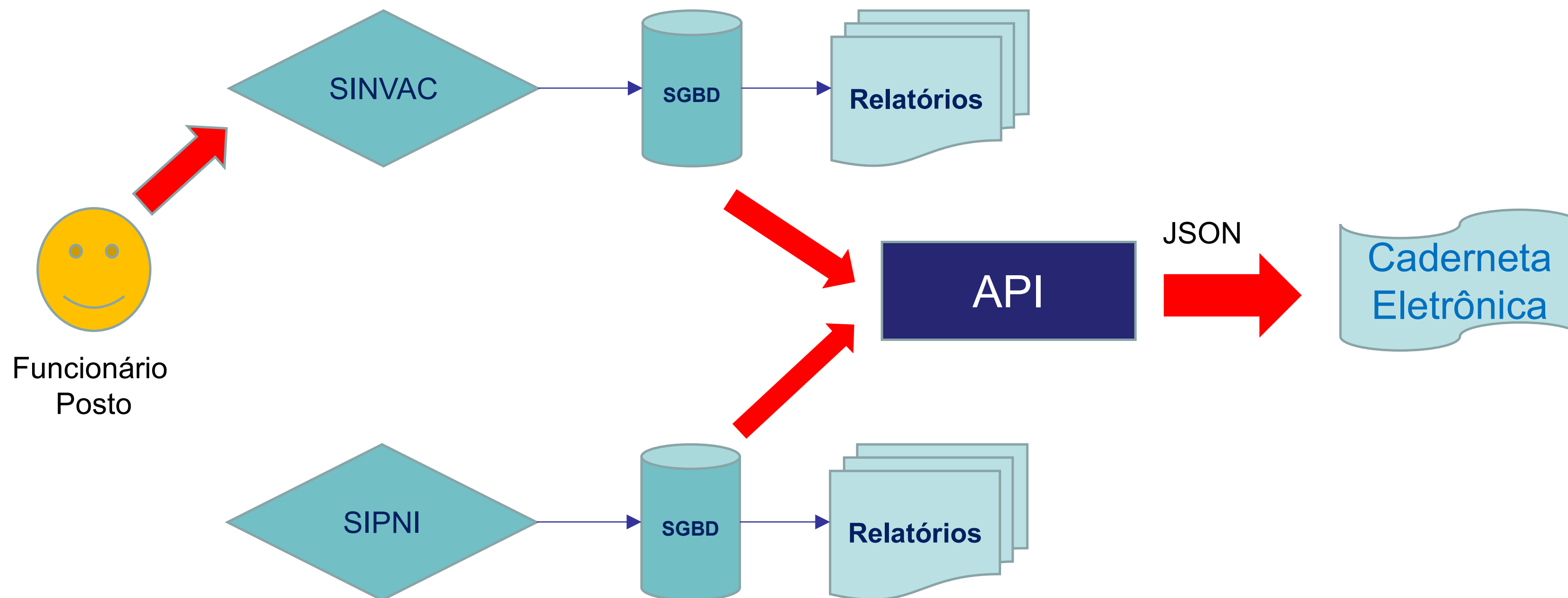
P(s): **se** s > 0 **então**
s = s-1;

s = 1-1 = 0

:: 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

API Rest



É IMPORTANTE FAZER UM DESENHO QUE MOSTRE AS INTERFACES DE COMUNICAÇÃO DA API
(de onde vem os dados e para onde eles vão)

API Rest

- Como os dados chegam na API? SGBD ➔ proposta
- A granularidade dos dados está adequada as necessidades da API? ➔ perguntas relacionadas ao desenvolvimento da aplicação (proposta)
- Porque usar esta API? Quais as vantagens? Que facilidades a API traz ao posto de saúde e/ou ao paciente? ➔ início (introdução), proposta
- Deixar claro o que você está fazendo: API é para o posto de saúde? API é para o paciente? O que é gerenciado a partir desta API?

FIGURA 2.28 O problema do produtor-consumidor usando semáforos.

Mutual exclusion

```
#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 */
RC              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 */
RC              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 */
        }
}
```

Comunicação de Processos (-- Semáforos --)

Como resolver o problema do Produtor vs. Consumidor usando semáforos?

Comunicação de Processos
(-- Problemas clássicos --)

- Jantar dos filósofos
- Escritores e Leitores
- Barbeiro dorminhoco

Comunicação entre processos (-- O jantar dos filósofos --)

Comunicação entre processos (-- O jantar dos filósofos --)

- Formulado por E. Dijkstra para caracterizar o problema da sincronização e concorrência

- Descrição
  - 5 filósofos numa mesa de jantar circular
  - 5 pratos de espaguete
  - 1 garfo entre cada par de pratos

Comunicação entre processos (-- O jantar dos filósofos --)

- Descrição
  - Cada filósofo pode "comer" ou "pensar"
  - Cada filósofo usa dois garfos para comer
  - Cada filósofo pega um garfo por vez

Jantar dos filósofos (-- 1ª solução --)

```
#define N 5

void philosopher (int i)
{
   while (TRUE)
   {
      think();
      take_fork (i);
      take_fork ((i+1) % N);
      eat();
      put_fork (i);
      put_fork ((i+1) % N);
   }
}
```

- O que acontece se todos os filósofos pegam o garfo da esquerda simultaneamente?
  – Nenhum filósofo consegue pegar o garfo da direita
  – ***DEADLOCK***

Jantar dos filósofos
(-- 2ª solução --)

```
#define N 5

void philosopher (int i)
{
   while (TRUE)
   {
      think();
      take_fork (i);
      if (fork((i+1) % N) is available)
      {
         take_fork ((i+1) % N);
         eat();
         put_fork (i);
         put_fork ((i+1) % N);
      }
      else
        put_fork (i);
   }
}
```

- O que acontece se todos os filósofos pegam o garfo da esquerda simultaneamente?
  - ***INANIÇÃO (starvation)***

Jantar dos filósofos (-- 3ª solução --)

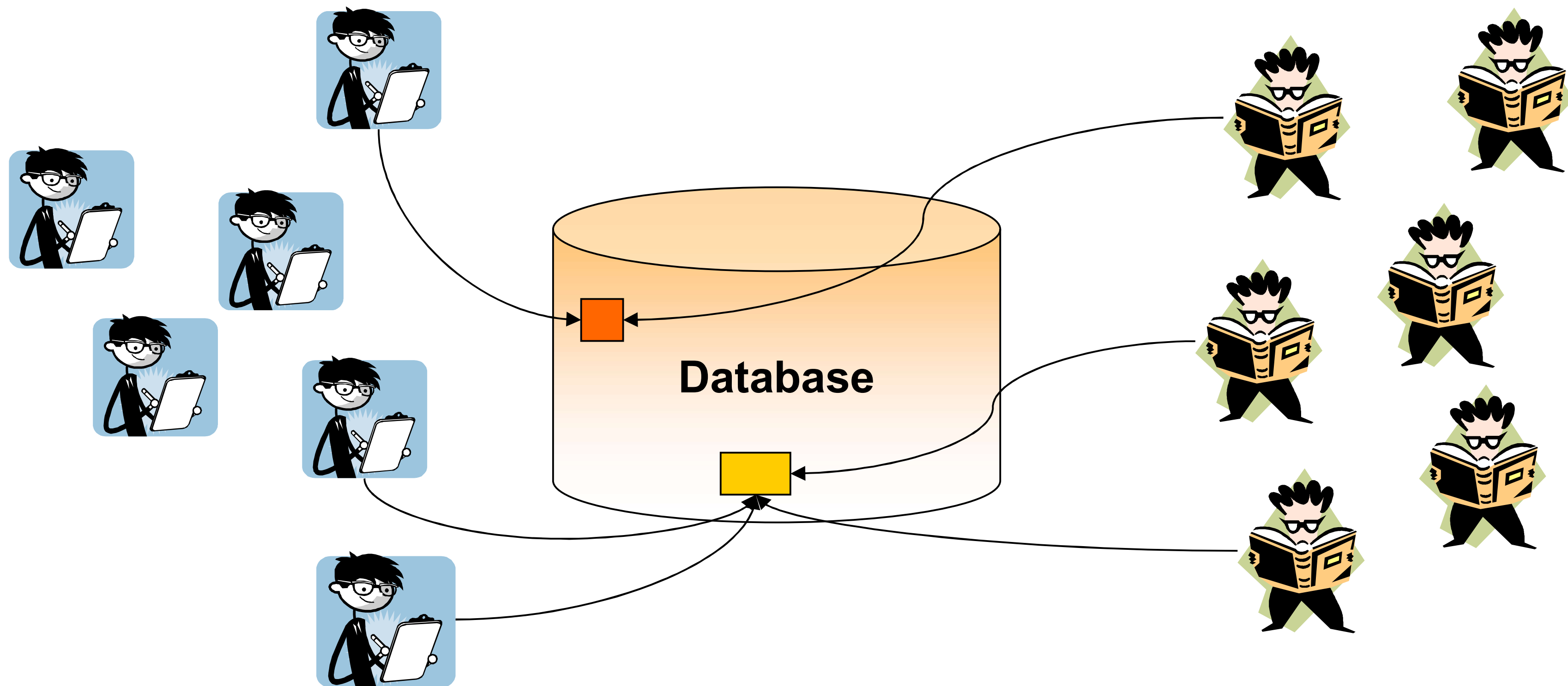
```
#define N 5

void philosopher (int i)
{
    while (TRUE)
    {
        think();
        down(mutex);
        take_fork (i);
        take_fork ((i+1) % N);
        eat();
        put_fork (i);
        put_fork ((i+1) % N);
        up(mutex);
    }
}
```

- O que acontece nesta solução?
  - Apenas um filósofo come por vez
  - Afeta o ***PARALELISMO***

Jantar dos filósofos (-- 3ª solução --)

- Atribui 3 possíveis estados aos filósofos
  - PENSANDO
  - COMENDO
  - FAMINTO
- Idéia:
  - Um filósofo no estado “faminto” só pode pegar os garfos se os seus vizinhos (esquerda e direita) não estiverem “comendo”.
- Estudar a solução para o problema dos filósofos!

Comunicação entre processos (-- Os leitores e escritores --)



Comunicação entre processos
(-- Barbeiro dorminhoco --)