

Sincronização de Processos (3)

Exercícios - Semáforos

Uso dos Semáforos

1. Sincronização de execução
2. Acesso a recursos limitados
3. Exclusão mútua
 - Problema do pombo correio
 - Problema do jantar dos canibais
 - Problema do filme sobre a vida de Hoare
4. Problemas clássicos
 - Leitores e escritores
 - Barbeiro dorminhoco
 - Jantar dos filósofos

Sincronização de Execução ⁽¹⁾

- Problema 1: Suponha que sejam criados 5 processos. Utilize semáforos para garantir que o processo 1 escreva os números de 1 até 200, o processo 2 escreva os números de 201 até 400, e assim por diante.
- Dica: o processo $i+1$ só deve entrar em funcionamento quando processo i já tiver terminado a escrita dos seus números.

Sincronização de Execução ⁽¹⁾

- Solução do problema 1

Semaphore S1, S2, S3, S4;

S1=S2=S3=S4=0

P0

...

Imprime os números de 1 até 200

UP(S1);

...

Termina

Pi

...

DOWN(Si);

...

Imprime números de $i*200+1$ até $(i+1)*200$

...

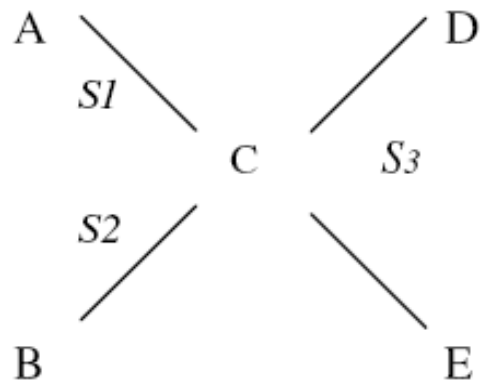
UP(Si+1)

...

Termina

Sincronização de Execução (2)

- Problema 2: Adicione semáforos ao programa abaixo, e as respectivas chamadas às suas operações, de modo que a precedência definida pelo grafo seja alcançada



semaphore ...

...

Process k

...

do some work k

...

end

/* Comentário */

/* Comentário */

Sincronização de Execução (2)

■ Solução do Problema 2

```
semaphore S1=0  
semaphore S2=0  
semaphore S3=0
```

```
Process A  
  //do some work  
  V(S1)
```

```
Process B  
  // do some work  
  V (S2)
```

```
Process C  
  P(S1)  
  P(S2)  
  //do some work  
  V(S3)  
  V(S3)
```

```
Process D  
  P(S3)  
  //do some work
```

```
Process E  
  P(S3)  
  //do some work
```

Problema do Pombo Correio (3)

- Problema 3: considere a seguinte situação.
 - Um pombo correio leva mensagens entre os sites A e B, mas só quando o número de mensagens acumuladas chega a 20.
 - Inicialmente, o pombo fica em A, esperando que existam 20 mensagens para carregar, e dormindo enquanto não houver.
 - Quando as mensagens chegam a 20, o pombo deve levar exatamente (nenhuma a mais nem a menos) 20 mensagens de A para B, e em seguida voltar para A.
 - Caso existam outras 20 mensagens, ele parte imediatamente; caso contrário, ele dorme de novo até que existam as 20 mensagens.
 - As mensagens são escritas em um post-it pelos usuários; cada usuário, quando tem uma mensagem pronta, cola sua mensagem na mochila do pombo. Caso o pombo tenha partido, ele deve esperar o seu retorno p/ colar a mensagem na mochila.
 - O vigésimo usuário deve acordar o pombo caso ele esteja dormindo.
 - Cada usuário tem seu bloquinho inesgotável de post-it e continuamente prepara uma mensagem e a leva ao pombo.
- Usando semáforos, modele o processo pombo e o processo usuário, lembrando que existem muitos usuários e apenas um pombo. Identifique regiões críticas na vida do usuário e do pombo.

Problema do Pombo Correio (3)

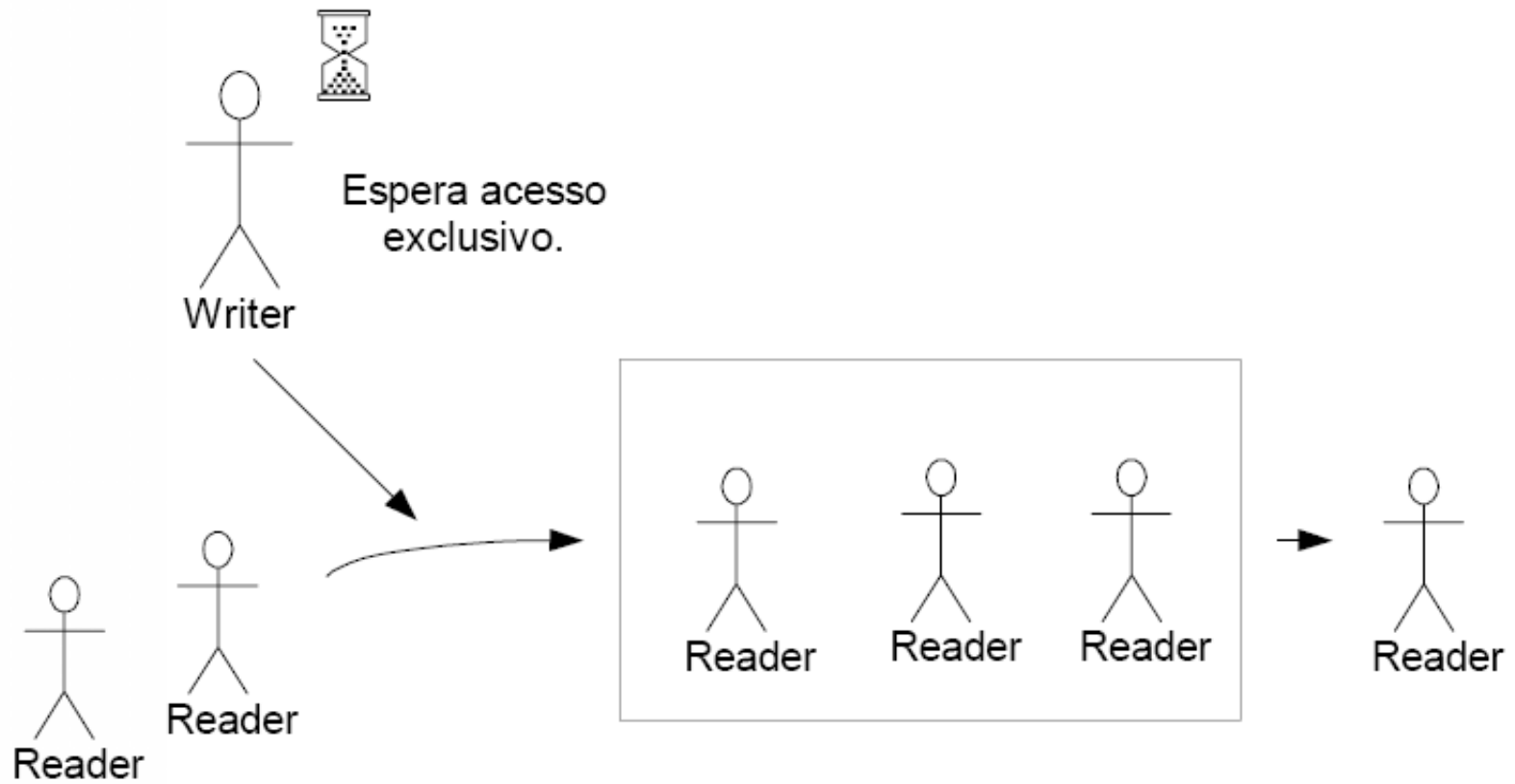
```
#define N=20
```

```
int contaPostIt=0;
semaforo mutex=1; //controlar acesso à variável contaPostIt
semaforo cheia=0; //usado para fazer o pombo dormir enquanto ã há 20 msg
semaforo enchendo=N; //Usado p/ fazer usuários dormirem enquanto pombo
                        //está fazendo o transporte
```

```
usuario() {
    while(true){
        down(enchendo);
        down(mutex);
        colaPostIt_na_mochila();
        contaPostIt++;
        if (contaPostIt==N)
            up(cheia);
        up(mutex);
    }
}
```

```
pombo() {
    while(true){
        down(cheia);
        down(mutex);
        leva_mochila_ate_B_e_volta();
        contaPostIt=0;
        for(i=0;i<N;i++)
            up(enchendo);
        up(mutex);
    }
}
```


Leitores e Escritores ⁽⁴⁾



Leitores e Escritores (4)

- Problema 4:
 - Suponha que existe um conjunto de processos que compartilham um determinado conjunto de dados (ex: um banco de dados).
 - Existem processos que lêem os dados
 - Existem processos que escrevem (gravam) os dados
- Análise do problema:
 - Se dois ou mais leitores acessarem os dados simultaneamente não há problemas
 - E se um escritor escrever sobre os dados?
 - Podem outros processos estarem acessando simultaneamente os mesmos dados?

Leitores e Escritores (prioridade dos leitores) (4)

- Os leitores podem ter acesso simultâneo aos dados compartilhados
- Os escritores podem apenas ter acesso exclusivo aos dados compartilhados

```
//número de leitores ativos
int rc

//protege o acesso à variável rc
Semaphore mutex

//Indica a um escritor se este
//pode ter acesso aos dados
Semaphore db

//Inicialização:
mutex=1,
db=1,
rc=0
```

```
Escritor
while (TRUE)
    down(db);
    //writing is
    //performed
    up(db);
    ...
```

```
Leitor
while (TRUE)
    down(mutex);
    rc++;
    if (rc == 1)
        down(db);
    up(mutex);
    //reading is
    //performed
    down(mutex);
    rc--;
    if (rc == 0)
        up(db);
    up(mutex);
```

Leitores e Escritores (prioridade dos leitores) (4)

```
typedef int semaphore;

semaphore mutex = 1;          /* controls access to rc */
semaphore db = 1;             /* controls access to the data base */
int rc = 0;                    /* no. of processes reading or wanting to read */

void reader(void)
{
    while (TRUE) {
        down(&mutex);          /* get exclusive access to rc */
        rc++;                  /* one reader more now */
        if (rc == 1) down(&db); /* if this is the first reader ... */
        up(&mutex);             /* release exclusive access to rc */
        read_data_base();       /* access the data */
        down(&mutex);          /* get exclusive access to rc */
        rc--;                  /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to rc */
        use_data();            /* non-critical section */
    }
}

void writer(void)
{
    while (TRUE) {
        prepare_data();         /* non-critical section */
        down(&db);              /* get exclusive access to data base */
        write_data_base();      /* update data base */
        up(&db);                /* release exclusive access to data base */
    }
}
```

Leitores e Escritores (prioridade dos escritores)

(5)

- Problema 5: a solução anterior pode levar à *starvation* dos escritores. A solução a seguir atende aos processos pela ordem de chegada, mas dando prioridade aos escritores
 - Dica: quando existir um escritor pronto para escrever, este tem prioridade sobre todos os outros leitores que chegarem depois dele.

```
rc          //Número de leitores
wc          //Número de escritores, apenas um escritor de cada vez pode ter acesso aos
           //dados compartilhados
mutex_rc    //Protege o acesso à variável rc
mutex_wc    //Protege o acesso à variável wc
mutex       //Impede que + do que 1 leitor tente entrar na região crítica
w_db        //Indica a um escritor se este pode ter acesso aos dados
r_db        //Permite que um processo leitor tente entrar na sua região crítica
```

rc	//Número de leitores	
wc	//Número de escritores, apenas um escritor de cada vez pode ter acesso aos dados compartilhados	
mutex_rc	// Protege o acesso à variável rc	Leitores e Escritores (prioridade dos escritores) (6)
mutex_wc	//Protege o acesso à variável wc	
mutex	//Impede que + do que 1 leitor tente entrar na região crítica	
w_db	//Indica a um escritor se este pode ter acesso aos dados	
r_db	//Permite que um processo leitor tente entrar na sua região crítica	

Inicialização

```
rc = 0
wc = 0

//semáforos
mutex_rc = 1
mutex_wc = 1
mutex = 1
w_db = 1
r_db = 1
```

Escritor

```
while (TRUE){
    down(mutex_wc);
    wc++;
    if (wc == 1)
        down(r_db);
    up(mutex_wc);
    down(w_db)

    //Escrita

    up(w_db)
    down(mutex_wc);
    wc--;
    if (wc == 0)
        up(r_db);
    up(mutex_wc);
}
```

Leitor

```
while (TRUE){
    down(mutex);
    down(r_db);
    down(mutex_rc);
    rc++;
    if (rc == 1)
        down(w_db);
    up(mutex_rc);
    up(r_db);
    up(mutex);

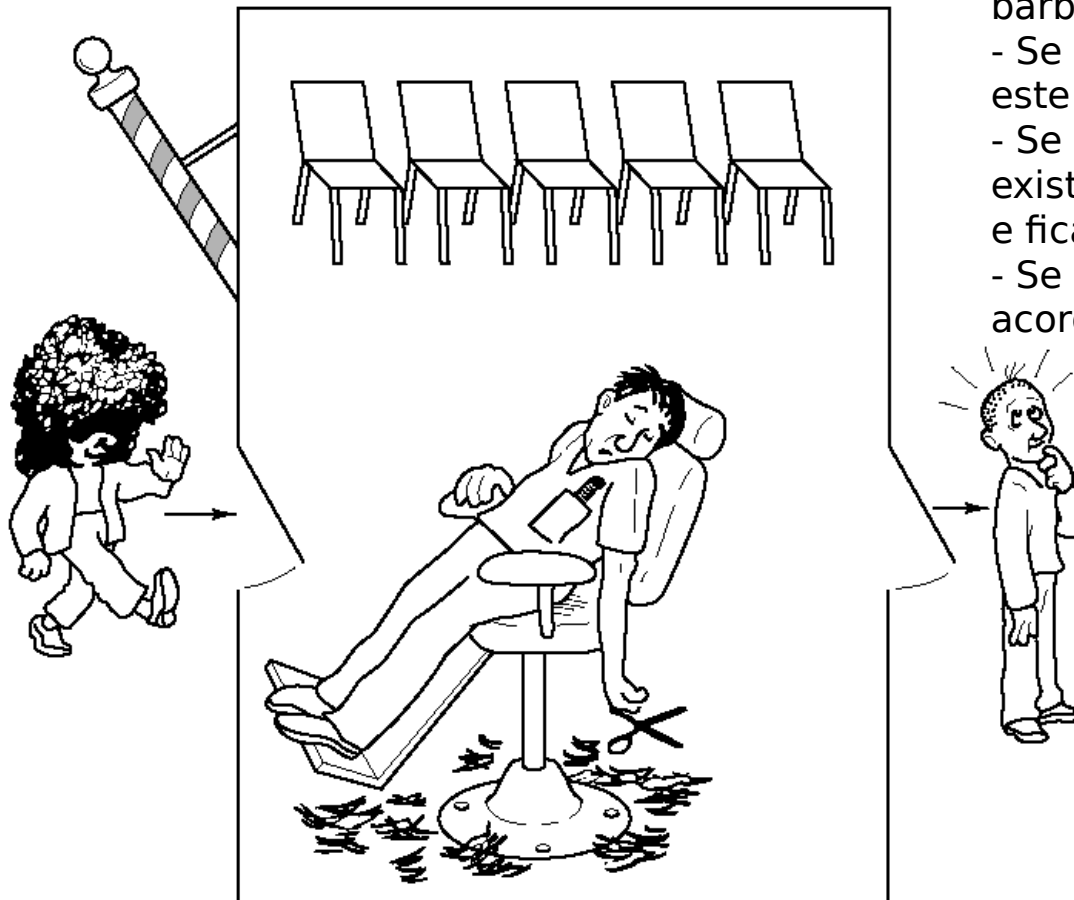
    //Leitura dos dados

    down(mutex_rc);
    rc--;
    if (rc == 0)
        up(w_db);
    up(mutex_rc); }
```

O Barbeiro Dorminhoco ⁽⁶⁾

Problema 6: a barbearia consiste numa sala de espera com n cadeiras mais a cadeira do barbeiro. Se não existirem clientes o barbeiro dorme. Ao chegar um cliente:

- Se todas as cadeiras estiverem ocupadas, este vai embora
- Se o barbeiro estiver ocupado, mas existirem cadeiras livres, o cliente senta-se e fica esperando sua vez
- Se o barbeiro estiver dormindo, o cliente o acorda e corta o cabelo.



```
#define CHAIRS 5
typedef int semaphore;
semaphore customers = 0;
semaphore barbers = 0;
semaphore mutex = 1;
int waiting = 0;
```

```
void barber(void) {
    while (TRUE)
    { down(customers);
      down(mutex);
      waiting = waiting - 1;
      up(barbers);
      up(mutex);
      cut_hair();
    } }
```

```
void customer(void) {
    down(mutex);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        up(customers);
        up(mutex);
        down(barbers);
        get_haircut(); }
    else {
        up(mutex); }}
```

```
/* # chairs for waiting customers */
/* use your imagination */
/* # of customers waiting for service */
/* # of barbers waiting for customers */
/* for mutual exclusion */
/* customers are waiting (not being cut) */
```

O Barbeiro Dorminhoco (6)

```
/* go to sleep if # of customers is 0 */
/* acquire access to 'waiting' */
/* decrement count of waiting customers */
/* one barber is now ready to cut hair */
/* release 'waiting' */
/* cut hair (outside critical region) */
```

```
/* enter critical region */
/* if there are no free chairs, leave */
/* increment count of waiting customers */
/* wake up barber if necessary */
/* release access to 'waiting' */
/* go to sleep if # of free barbers is 0 */
/* be seated and be serviced */

/* shop is full; do not wait */
```


OS Filósofos Glutões (7)

Problema 7: considere cinco filósofos que passam a vida a comer e a pensar. Eles compartilham uma mesa circular, com um prato de arroz ao centro. Na mesa existem cinco pauzinhos, colocados um de cada lado do filósofo.

Quando um filósofo fica com fome ele pega os dois pauzinhos mais próximos, um de cada vez, e come até ficar saciado. Quando acaba de comer, ele repousa os pauzinhos e volta a pensar.



OS Filósofos Glutões (7)

Dados
Compartilhados

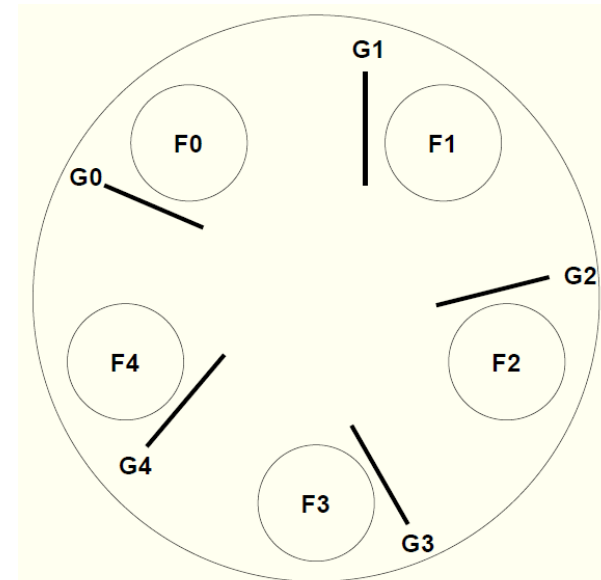
```
typedef int semaphore;  
  
semaphore forks[N] =  
{1,1,...,1};
```

```
void philosopher(int i)  
{ while (TRUE) {  
  think();  
  down(forks[i]);  
  down(forks[i+1]);  
  eat();  
  up(forks[i]);  
  up(forks[i+1]);  
}
```

Solução simples (garante que dois vizinhos ã
comam simultaneamente) ... mas ela pode
causar deadlock!!!



Para resolver isso, uma solução seria permitir
que um filósofo pegue seus garfos somente se
ambos estiverem livres (neste caso isso deve
ser feito dentro de uma região crítica...)



OS Filósofos Glutões (7) – Outra Tentativa

Dados

Compartilhados

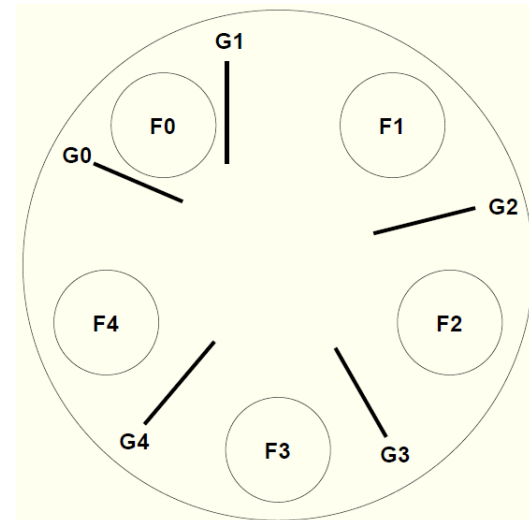
```
typedef int semaphore;
```

```
semaphore forks[N] =  
{1,1,...,1};
```

```
semaphore mutex = 1;
```

```
void philosopher(int i)  
{ while (TRUE) {  
  think();  
  down(mutex);  
  down(forks[i]);  
  down(forks[i+1])  
  eat();  
  up(forks[i]);  
  up(forks[i+1]);  
  up(mutex);  
}
```

Baixo DESEMPENHO : sem paralelismo !



OS Filósofos Glutões

(7) – Solução –

```
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N
#define THINKING 0
#define HUNGRY 1
#define EATING 2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];
```

```
void philosopher(int i)
{ while (TRUE) {
    think();
    take_forks(i);
    eat();
    put_forks(i); }}
```

```
void take_forks(int i)
{ down(&mutex);
  state[i] = HUNGRY;
  test(i);
  up(&mutex);
  down(&s[i]); }
```

```
void put_forks(i)
{down(&mutex);
 state[i] = THINKING;
 test(LEFT);
 test(RIGHT);
 up(&mutex); }
```

```
void test(i)
{ if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
    state[i] = EATING; up(&s[i]); }}
```

```
/* number of philosophers */
/* number of i's left neighbor */
/* number of i's right neighbor */
/* philosopher is thinking */
/* philosopher is trying to get forks */
/* philosopher is eating */
/* semaphores are a special kind of int */
/* array to keep track of everyone's state */
/* mutual exclusion for critical regions */
/* one semaphore per philosopher */
```

```
/* i: philosopher number, from 0 to N-1 */
/* repeat forever */
/* philosopher is thinking */
/* acquire two forks or block */
/* yum-yum, spaghetti */
/* put both forks back on table */
```

```
/* i: philosopher number, from 0 to N-1 */
/* enter critical region */
/* record fact that philosopher i is hungry */
/* try to acquire 2 forks */
/* exit critical region */
/* block if forks were not acquired */
```

```
/* i: philosopher number, from 0 to N-1 */
/* enter critical region */
/* philosopher has finished eating */
/* see if left neighbor can now eat */
/* see if right neighbor can now eat */
/* exit critical region */
```

```
/* i: philosopher number, from 0 to N-1 */
```