Laboratório de Pesquisa em Redes e Multimídia

Sincronização de Processos (3)

Exercícios - Semáforos

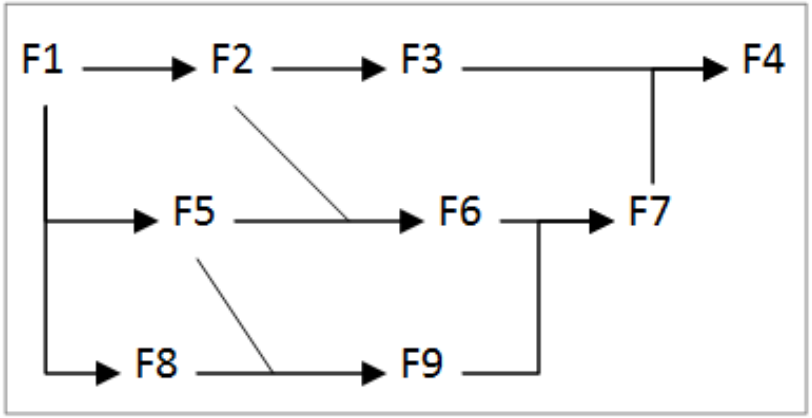
Universidade Federal do Espírito Santo
Departamento de Informática

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
1. 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.
- **Problema 2**: Considere o seguinte grafo de precedência que será executado por três processos. Adicione semáforos a este programa (no máximo 6 semáforos), e as respectivas chamadas às suas operações, de modo que a precedência definida abaixo seja alcançada.




```
PROCESS A : begin F1 ; F2 ; F9 ; F4 ; end

PROCESS B : begin F3 ; F7 ; end

PROCESS C : begin F8 ; F5 ; F6 ; end
```

Sincronização de Execução (2)

- Solução do problema 1

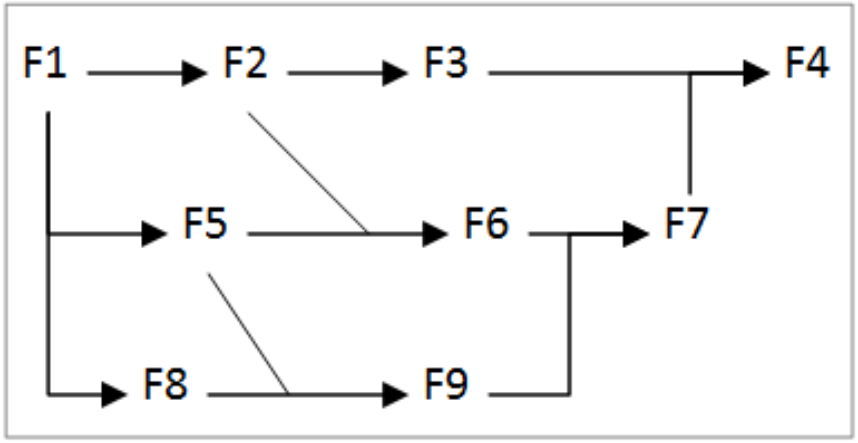
```
Semaphore S1, S2, S3, S4;
S1=S2=S3=S4=0
```

```
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 (3)

- Solução do problema 2




```
PROCESS A : begin F1 ; F2 ; F9 ; F4 ; end

PROCESS B : begin F3 ; F7 ; end

PROCESS C : begin F8 ; F5 ; F6 ; end
```

Solução não otimizada:

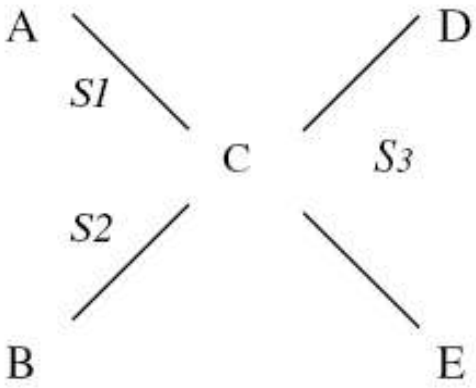
```
PROCESS A : begin F1 ; V(s1); F2 ; V(s2); V(s3); P(s4); F9 ; V(s3); P(s5); F4 ; end

PROCESS B : begin P(s2); F3 ; P(s3); P(s3); F7 ; V(s5); end

PROCESS C : begin P(s1); F8 ; F5 ; P(s3); V(s4); F6 ; V(s3); end
```

Sincronização de Execução (4)

- Problema 3: 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
        ...                         /* Comentário */
        do some work k
        ...                         /* Comentário */
end
```

Sincronização de Execução (5)

- Solução do Problema 3

```
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 (1)

- 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 (2)

```
#define N=20

int contaPostIt=0;
semaforo mutex=1; //controlar acesso à variável contaPostIt
semaforo cheia=0; //usado para fazer o pombo dormir enquanto ñ 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);
 }
}
```

Problema do Carrinho da Montanha Russa (1)

- Considere a seguinte situação:
  - *"Existem n passageiros que, repetidamente, aguardam para entrar em um carrinho da montanha russa, fazem o passeio e voltam a aguardar. Vários passageiros podem entrar no carrinho ao mesmo tempo, pois este tem várias portas. A montanha russa tem somente um carrinho, onde cabem C passageiros (C < n). O carrinho só começa seu percurso se estiver lotado."*
- Resolva esse problema usando semáforos.

Problema do Jantar dos Canibais (1)

Suponha que um grupo de N canibais come jantares a partir de uma grande travessa que comporta M porções. Quando alguém quer comer, ele(ela) se serve da travessa, a menos que ela esteja vazia. Se a travessa está vazia, o canibal acorda o cozinheiro e espera até que o cozinheiro coloque mais M porções na travessa.

Desenvolva o código para as ações dos canibais e do cozinheiro. A solução deve evitar *deadlock* e deve acordar o cozinheiro apenas quando a travessa estiver vazia. Suponha um longo jantar, onde cada canibal continuamente se serve e come, sem se preocupar com as demais coisas na vida de um canibal...

Problema do Jantar dos Canibais (2)

```
semaphore cozinha = 0
semaphore comida = M+1
semaphore mutex = 1
semaphore enchendo = 0
int count = 0
```

Canibal

```
While (1) {
   P(comida)
   P(mutex)
   count++
   if (count > M) then
      V(cozinha)
      P(enchendo)
      count=1
   V(mutex);
   come();
}
```

Cozinheiro

```
While (1) {
   P(cozinha)
   enche_travessa()
   for (int i=1; i ≤ M; i++)
       V(comida);
   V(enchendo)
}
```

Problema do Filme sobre a Vida de Hoare (1)

Em um determinado stand de uma feira, um demonstrador apresenta um filme sobre a vida de Hoare. Quando 10 pessoas chegam, o demonstrador fecha o pequeno auditório que não comporta mais do que essa platéia. Novos candidatos a assistirem o filme devem esperar a próxima exibição.

Esse filme faz muito sucesso com um grupo grande de fãs (de bem mais de 10 pessoas), que permanecem na feira só assistindo o filme seguidas vezes. Cada vez que um **desses fãs consegue assistir uma vez o filme por completo**, ele vai telefonar para casa para contar alguns detalhes novos para a sua mãe. Depois de telefonar ele volta mais uma vez ao stand para assistir o filme outra vez.

Usando semáforos, modele o processo fã e o processo demonstrador, lembrando que existem muitos fãs e apenas um demonstrador. Como cada fã é muito ardoroso, uma vez que ele chega ao stand ele não sai dali até assistir o filme.

Suponha que haja muitos telefones disponíveis na feira e, portanto, que a tarefa de telefonar para casa não impõe nenhuma necessidade de sincronização.

Problema do Filme sobre a Vida de Hoare (2)

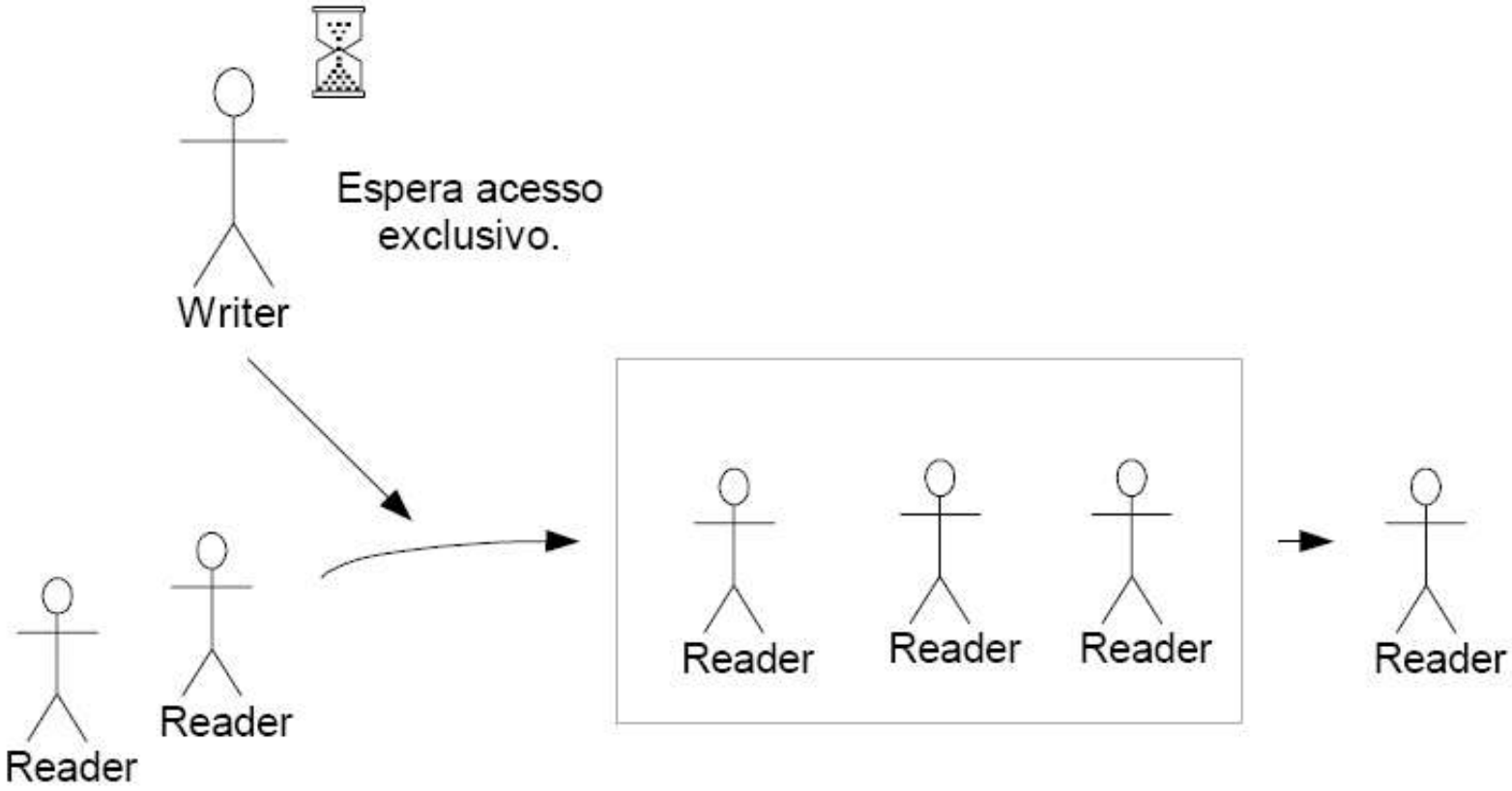
```
#define N 10
int nFans=0;
semaphore mutex = 1;
semaphore dem = 0;  //usado p/ bloquear o dem.
semaphore fila = 0; // usado p/ bloquear as
pessoas
```

```
fan (){
  while(true){
    P(mutex);
    nFans++;
    V(mutex);
    V(dem) ;
    P(fila) ;
    assisteFilme() ;
    telefona();
  }
}
```

```
demonstrador (){
  while(true){
    while (nFans<N)
      P(dem);
    P(mutex);
    nFans=nFans-N;
    V(mutex);
    for (i=0 ;i<N ; i++)
      V(fila) ;
    exibeFilme() ;
  }
}
```

Qual o problema desta solução ???

Leitores e Escritores (1)



Leitores e Escritores (2)

- Problema:
  - 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) (3)

- 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

//Incialização:
    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 escritores) (5)

- A solução anterior pode levar a *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
```

Leitores e Escritores (prioridade dos escritores) (6)

```
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
```

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

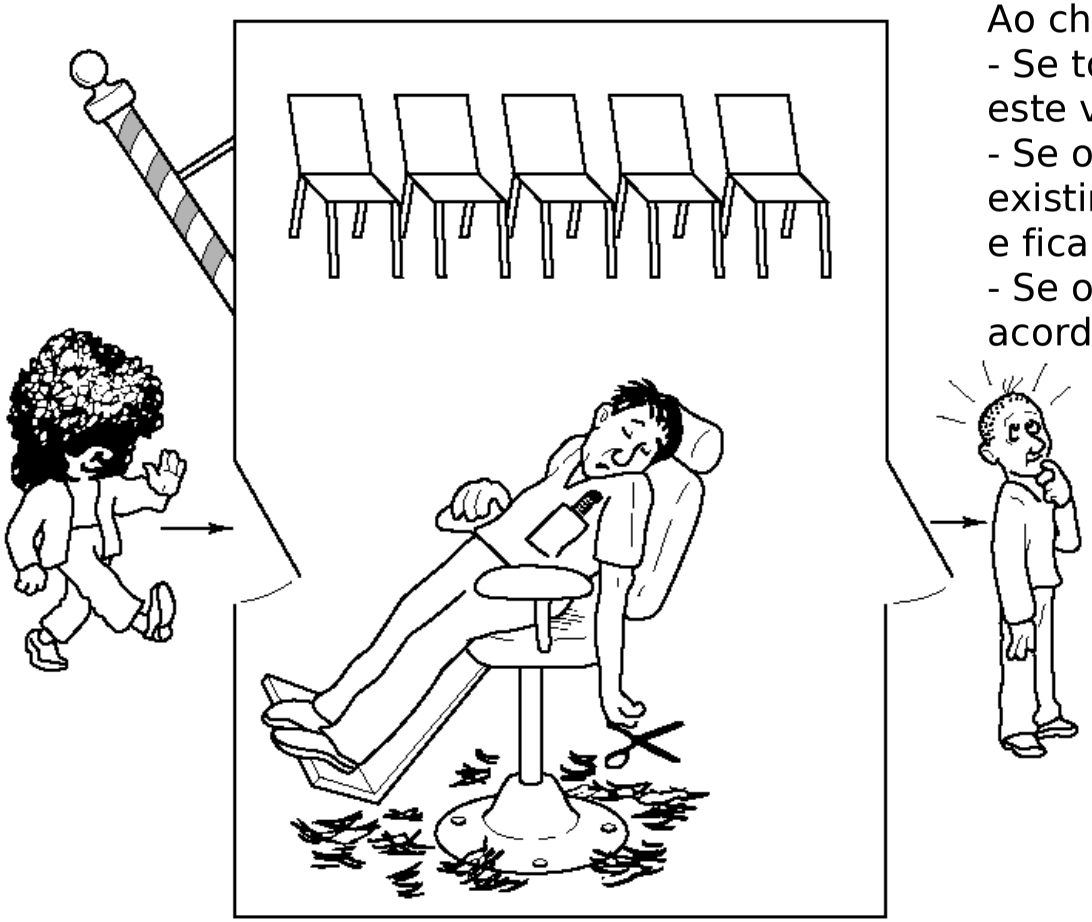
```
while (TRUE){
  down(mutex);
  down(r_db);
  down(mutex_rc);
  rc++;
  if (rc == 1)
    down(w_db);
  up(mutex_rc);
  up(r_db);
  up(mutex);
  …
  //Leitura dos dados
  …
  down(mutex_rc);
  rc--;
  if (rc == 0)
    up(w_db);
  up(mutex_rc); )
```

- Problema dos Leitores e Escritores
  - Esta solução pode levar a *starvation* de escritores?
  - Todos os semáforos são iniciados com valor 1

```
leitor:
...
1 while(1) {
2 P(R);
3 P(M);
4 rc++;
5 if (rc==1) P(W);
6 V(M);
7 V(R);
8 LE;
9 P(M);
10 rc--;
11 if (rc==0) V(W);
12 V(M);
13 ... consome
```

```
escritor:
...
1 while(1) {
2 ...produz
3 P(R);
4 P(W);
5 ESCREVE;
6 V(W);
7 V(R);
8 }
```

O Barbeiro Dorminhoco (1)



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.

O Barbeiro Dorminhoco (2)

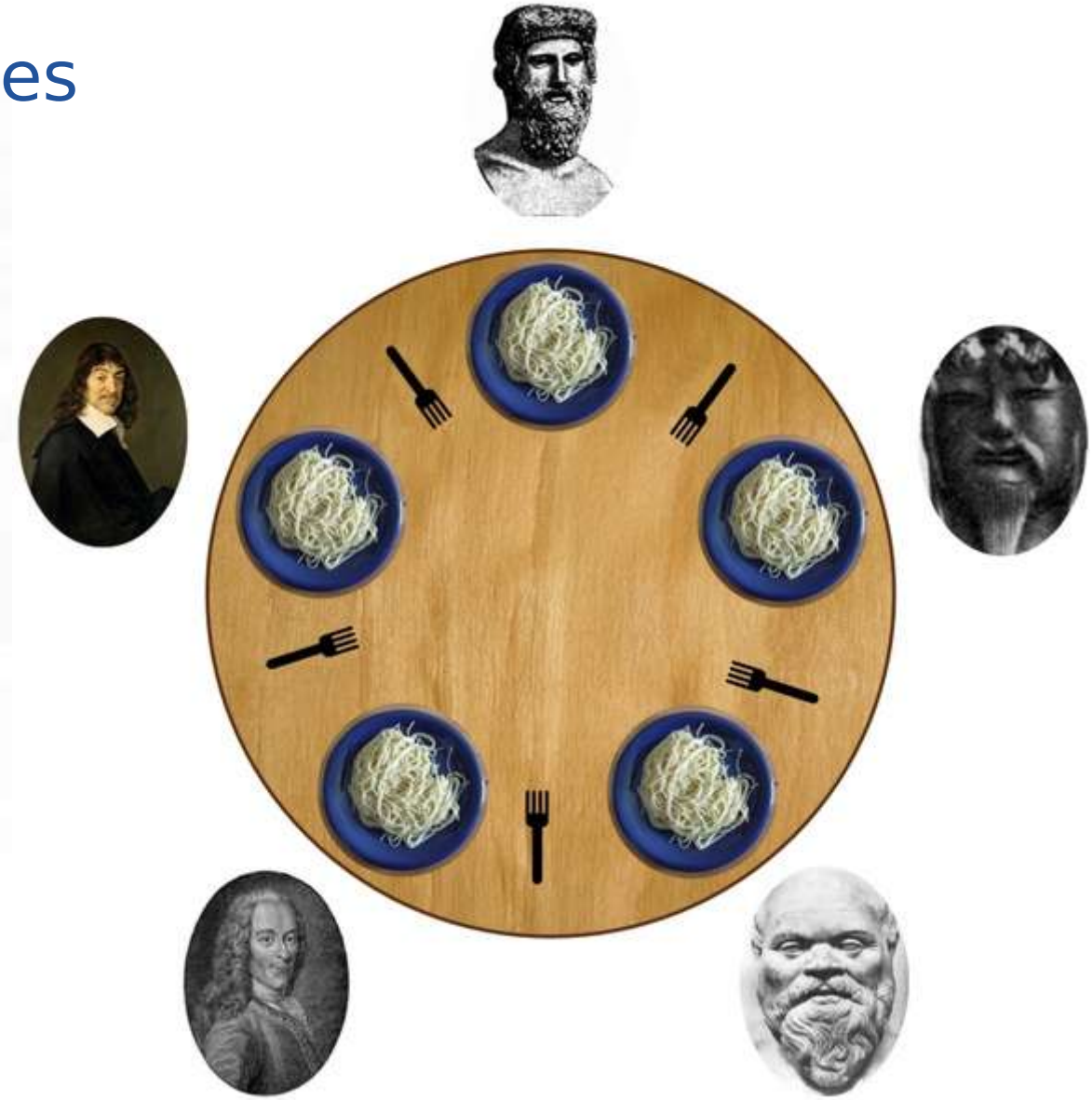
```
#define CHAIRS 5                    /* # chairs for waiting customers */
typedef int semaphore;              /* use your imagination */
semaphore customers = 0;            /* # of customers waiting for service */
semaphore barbers = 0;               /* # of barbers waiting for customers */
semaphore mutex = 1;                /* for mutual exclusion */
int waiting = 0;                    /* customers are waiting (not being cut) */

void barber(void) {
   while (TRUE)
   { down(customers);               /* go to sleep if # of customers is 0 */
      down(mutex);                  /* acquire access to 'waiting' */
      waiting = waiting - 1;        /* decrement count of waiting customers */
      up(barbers);                  /* one barber is now ready to cut hair */
      up(mutex);                     /* release 'waiting' */
      cut_hair();                   /* cut hair (outside critical region) */
    } }

void customer(void) {
   down(mutex);                     /* enter critical region */
   if (waiting < CHAIRS) {          /* if there are no free chairs, leave */
      waiting = waiting + 1;        /* increment count of waiting customers */
      up(customers);                /* wake up barber if necessary */
      up(mutex);                    /* release access to 'waiting' */
      down(barbers);                /* go to sleep if # of free barbers is 0 */
      get_haircut();  }             /* be seated and be serviced */
   else {
     up(mutex); }}                  /* shop is full; do not wait */
```

OS Filósofos Glutões

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 (2) - Solução

Dados Compartilhados

```
#define N 5
#define LEFT (i+N-1)%N
#define RIGHT (i+1)%N

#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];

typedef int semaphore;

semaphore mutex = 1;
semaphore philo[N]=
  {0,0,...,0};
```

```
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(&philo[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(&philo[i]);
  }
}
```

OS Filósofos Glutões
(3) – Solução –

```
#define N 5                                     /* number of philosophers */
#define LEFT (i+N-1)%N                          /* number of i's left neighbor */
#define RIGHT (i+1)%N                           /* number of i's right neighbor */
#define THINKING 0                              /* philosopher is thinking */
#define HUNGRY 1                                /* philosopher is trying to get forks */
#define EATING 2                                /* philosopher is eating */
typedef int semaphore;                          /* semaphores are a special kind of int */
int state[N];                                   /* array to keep track of everyone's state */
semaphore mutex = 1;                            /* mutual exclusion for critical regions */
semaphore s[N];                                 /* one semaphore per philosopher */

void philosopher(int i)                         /* i: philosopher number, from 0 to N-1 */
   { while (TRUE) {                             /* repeat forever */
      think();                                  /* philosopher is thinking */
      take_forks(i);                            /* acquire two forks or block */
      eat();                                    /* yum-yum, spaghetti */
      put_forks(i); }}                          /* put both forks back on table */

void take_forks(int i)                          /* i: philosopher number, from 0 to N-1 */
     { down(&mutex);                            /* enter critical region */
     state[i] = HUNGRY;                         /* record fact that philosopher i is hungry */
     test(i);                                   /* try to acquire 2 forks */
     up(&mutex);                                /* exit critical region */
     down(&s[i]); }                             /* block if forks were not acquired */

void put_forks(i)                               /* i: philosopher number, from 0 to N-1 */
     {down(&mutex);                             /* enter critical region */
      state[i] = THINKING;                      /* philosopher has finished eating */
      test(LEFT);                               /* see if left neighbor can now eat */
      test(RIGHT);                              /* see if right neighbor can now eat */
      up(&mutex); }                             /* exit critical region */

void test(i)                                    /* i: philosopher number, from 0 to N-1 */
{ if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
      state[i] = EATING; up(&s[i]); }}
```

Referências

- A. S. Tanenbaum, ''Sistemas Operacionais Modernos'', 3a. Edição, Editora Prentice-Hall, 2010.
  - Seção 2.3
- Silberschatz A. G.; Galvin P. B.; Gagne G.; ''Fundamentos de Sistemas Operacionais'', 8a. Edição, Editora LTC, 2010.
  - Seções 6.5 e 6.6
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; “Sistemas Operacionais”, 3ª. Edição, Editora Prentice-Hall, 2005
  - Seção 5.6