Laboratório de Pesquisa em Redes e Multimídia

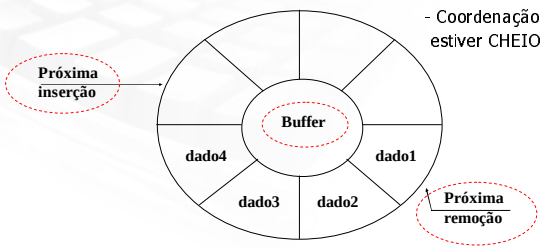
Sincronização de Processos (4)

Aula 13 - Exercícios - Semáforos

Universidade Federal do Espírito Santo
Departamento de Informática

Produtor - Consumidor c/ *Buffer* Circular (1)

Dois problemas p/ resolver:
- Variáveis compartilhada
- Coordenação quando o buffer estiver CHEIO ou VAZIO



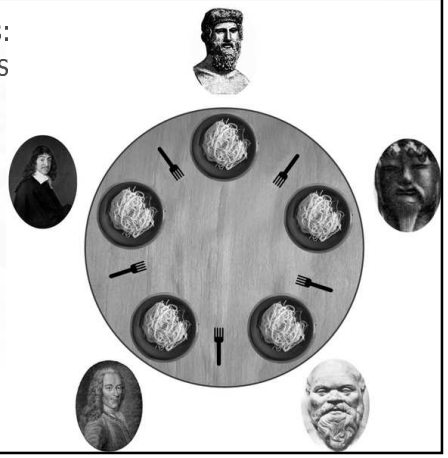
Produtor Consumidor c/ *Buffer* Circular (2)

- *Buffer* com capacidade N (vetor de N elementos).
- Variáveis *proxima_insercao* e *proxima_remocao* indicam onde deve ser feita a próxima inserção e remoção no *buffer*.
- Efeito de *buffer* circular é obtido através da forma como essas variáveis são incrementadas. Após o valor *N-1* elas voltam a apontar para a entrada zero do vetor
  - % representa a operação "resto da divisão"
- Três semáforos, duas funções diferentes: exclusão mútua e sincronização.
  - *mutex*: garante a exclusão mútua. Deve ser iniciado com "1".
  - *espera_dado*: bloqueia o consumidor se o *buffer* estiver vazio. Iniciado com "0".
  - *espera_vaga*: bloqueia produtor se o *buffer* estiver cheio. Iniciado com "N".

Produtor - Consumidor c/ *Buffer* Circular (3)

```
struct tipo_dado buffer[N];
int proxima_insercao = 0;
int proxima_remocao = 0;
...
semaphore mutex = 1;
semaphore espera_vaga = N;
semaphore espera_dado = 0;

----------------------------------------
void produtor(void){
...
down(espera_vaga);
down(mutex);
buffer[proxima_insercao] = dado_produzido;
proxima_insercao = (proxima_insercao + 1) % N;
up(mutex);
up(espera_dado);
... }

----------------------------------------
void consumidor(void){
...
down(espera_dado);
down(mutex);
dado_a_consumir := buffer[proxima_remocao];
proxima_remocao := (proxima_remocao + 1) % N;
up(mutex);
up(espera_vaga);
... }
```

Problemas Clássicos: OS Filósofos Glutões



Problemas Clássicos: OS Filósofos Glutões

Dados Compartilhados

```
typedef int semaphore;

semaphore forks[N];
```

```
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 ñ 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...)

Problemas Clássicos: OS Filósofos Glutões

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 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]);
  }
}
```

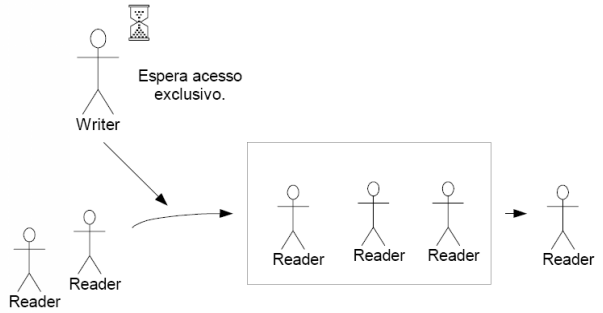

Solução para OS Filósofos Glutões

```
#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]); }}
```

Problemas Clássicos: Leitores e Escritores

- 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 (mas não os apagam!)
  - Existem processos que escrevem 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?

Problemas Clássicos: Leitores e Escritores



Leitores e Escritores: Solução possível (1)

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

```
//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);
```


Problemas Clássicos: Leitores e Escritores

```
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: Solução possível (2)

- A solução proposta é simples mas pode levar à starvation do escritor
- Exercício
  - Desenvolver uma solução que atenda os 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 cheguem após ele querendo ler os dados

```
rcount   //Número de leitores
wcount   //Número de escritores, apenas um escritor de cada vez pode ter acesso aos
         //dados compartilhados
mutex_rcount      // Protege o acesso à variável rcount
mutex_wcount      //Protege o acesso à variável wcount
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
```



```
rcount   //Número de leitores
wcount   //Número de escritores, apenas um escritor de cada vez pode ter acesso aos
         //dados compartilhados
mutex_rcount      // Protege o acesso à variável rc
mutex_wcount      //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
 rcount = 0
 wcount = 0

 //semáforos
 mutex_rcount = 1
 mutex_wcount = 1
 mutex = 1
 w_db = 1
 r_db = 1
```

```
Escritor
while (TRUE){
  down(mutex_wcount);
  wcount++;
  if (wcount == 1)
    down(r_db);
  up(mutex_wcount);
  down(w_db)
  …
  //Escrita
  …
  up(w_db)
  down(mutex_wcount);
  wcount--;
  if (wcount == 0)
    up(r_db);
  up(mutex_wcount);
}
```

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


Problema do Pombo correio

- 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 e não críticas na vida do usuário e do pombo.

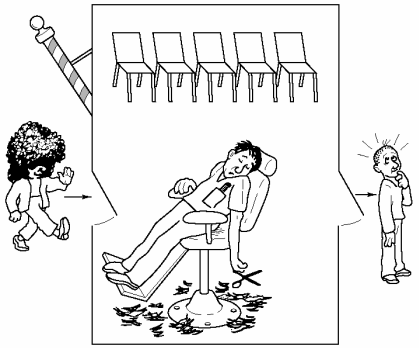
Problema do Pombo correio: Solução

```
#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);
 }
}
```

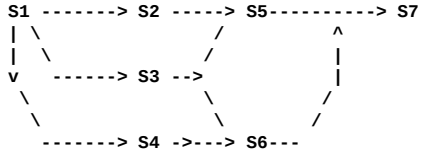
Problemas Clássicos: O Barbeiro Dorminhoco




```
#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 */
```


Exercício

- Considere o seguinte grafo de precedência



que será executado por três processos, conforme código abaixo:

```
cobegin
      P1: begin S1; S3; end;
      P2: begin S2; S5; S7; end;
      P3: begin S4; S6; end;
coend
```

Adicione semáforos a este programa, e as respectivas chamadas às suas operações, de modo que a precedência definida acima seja alcançada.

Exercício (2)

- Um *semáforo múltiplo* estende semáforos de maneira a permitir que as primitivas *up* e *down* operem em vários semáforos simultaneamente. Isto é útil para requisitar e liberar vários recursos através de uma operação atômica. Como você implementaria tais primitivas usando os semáforos estudados no curso? (Observe que você, como implementador da primitiva, pode acessar o valor do contador associado ao semáforo diretamente. Ou seja, você tem disponível na hora de implementar Down (R1, R2, ..., Rn) ou Up (R1, R2, ..., Rn) uma rotina "valor_do_contador (R)", que devolve o valor do contador associado a um semáforo).

Resposta

```
Down (R1, R2, ... , Rn) {
  while (1){
    down(mutex1); //garante exclusão mútua
    if ((valor_do_contador(R1)>0) &&
       (valor_do_contador(R2)>0) && ...
       (valor_do_contador(Rn)>0))
        break;  //sai do loop
        down(mutex2); //garante atomicidade dos multiplos down's
        up(mutex1);
     else {
      up(mutex1);//Libera o semáforo
      sleep(m);// e adormece por um tempo m p/ tentar fazer um down mais tarde
    }
  }
  down(R1);
  down(R2);
  ...
  down(Rn);
  up(mutex2);
}
```


Resposta

```
Up (R1, R2, ... , Rn) {
  down(mutex);
  up(R1);
  up(R2);
  ...
  up(Rn);
  up(mutex);
}
```