URI | UNIVERSIDADE REGIONAL INTEGRADA DO ALTO URUGUAI E DAS MISSÕES

30-769

Linguagem de Programação IV

MSc. Fernando Schubert

PROBLEMAS CLÁSSICOS

- Produtor-consumidor com buffer limitado
- Jantar dos filósofos
- Barbeiro dorminhoco
- Leitores e escritores

PRODUTOR-CONSUMIDOR COM BUFFER LIMITADO

- Este problema consiste em coordenar o acesso de tarefas (processos ou threads) a um buffer compartilhado com capacidade de armazenamento limitada a N itens (que podem ser inteiros, registros, mensagens, etc.). São considerados dois tipos de processos com comportamentos cíclicos e simétricos:[
  - **Produtor**: produz e deposita um item no buffer, caso haja uma vaga disponível. Caso contrário, deve esperar até que uma vaga seja liberada. Ao depositar um item, o produtor "consome" uma vaga livre.
  - **Consumidor**: retira um item do buffer e o consome; se o buffer estiver vazio, aguarda que novos itens sejam depositados pelos produtores. Ao consumir um item, o consumidor "produz" uma vaga livre no buffer.
- Deve-se observar que o acesso ao buffer é bloqueante, ou seja, cada processo fica bloqueado até conseguir realizar seu acesso, seja para produzir ou para consumir um item.

PRODUTOR-CONSUMIDOR COM BUFFER LIMITADO

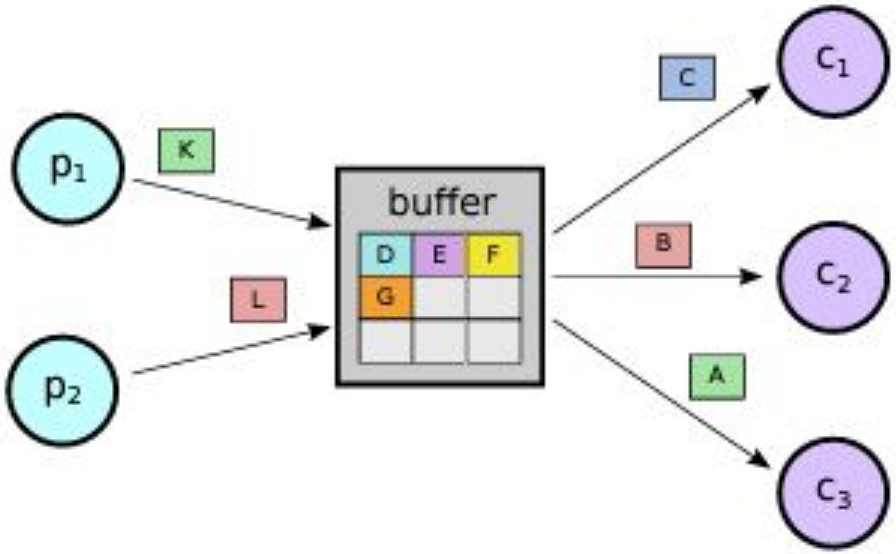
- 

Figura 12.1: O problema dos produtores/consumidores.

PRODUTOR-CONSUMIDOR COM BUFFER LIMITADO

- A solução para o problema dos produtores/consumidores envolve três aspectos de coordenação distintos e complementares:
  - A exclusão mútua no acesso ao buffer, para evitar condições de disputa entre produtores e/ou consumidores que poderiam corromper o conteúdo do buffer.
  - A suspensão dos produtores no caso de o buffer estar cheio: os produtores devem esperar até que surjam vagas livres no buffer.
  - A suspensão dos consumidores no caso de o buffer estar vazio: os consumidores devem esperar até que surjam novos itens a serem consumidos no buffer.

PRODUTOR-CONSUMIDOR COM BUFFER LIMITADO

- Solução usando semáforos:
  - Pode-se resolver o problema dos produtores/consumidores de forma eficiente utilizando um mutex e dois semáforos, um para cada aspecto de coordenação envolvido.

PRODUTOR-CONSUMIDOR COM BUFFER LIMITADO

```
1  mutex mbuf ;          // controla o acesso ao buffer
2  semaphore item ;      // controla os itens no buffer (inicia em 0)
3  semaphore vaga ;      // controla as vagas no buffer (inicia em N)
4
5  task produtor ()
6  {
7     while (1)
8     {
9        ...                    // produz um item
10       down (vaga) ;          // espera uma vaga no buffer
11       lock (mbuf) ;          // espera acesso exclusivo ao buffer
12       ...                    // deposita o item no buffer
13       unlock (mbuf) ;        // libera o acesso ao buffer
14       up (item) ;            // indica a presença de um novo item no buffer
15    }
16 }
17
18 task consumidor ()
19 {
20    while (1)
21    {
22       down (item) ;          // espera um novo item no buffer
23       lock (mbuf) ;          // espera acesso exclusivo ao buffer
24       ...                    // retira o item do buffer
25       unlock (mbuf) ;        // libera o acesso ao buffer
26       up (vaga) ;            // indica a liberação de uma vaga no buffer
27       ...                    // consome o item retirado do buffer
28    }
29 }
```

PRODUTOR-CONSUMIDOR COM BUFFER LIMITADO

- Solução usando variáveis de condição:
  - O problema dos produtores/consumidores também pode ser resolvido com variáveis de condição. Além do mutex para acesso exclusivo ao buffer, são necessárias variáveis de condição para indicar a presença de itens e de vagas no buffer. A listagem a seguir ilustra uma solução, lembrando que **N** é a capacidade do buffer e **num_itens** é o número de itens no buffer em um dado instante.

PRODUTOR-CONSUMIDOR COM BUFFER LIMITADO

```
1  mutex mbuf ;                      // controla o acesso ao buffer
2  condition item ;                  // condição: existe item no buffer
3  condition vaga ;                  // condição: existe vaga no buffer
4
5  task produtor ()
6  {
7     while (1)
8     {
9        ...                         // produz um item
10       lock (mbuf) ;               // obtem o mutex do buffer
11       while (num_itens == N)      // enquanto o buffer estiver cheio
12          wait (vaga, mbuf) ;      // espera uma vaga, liberando o buffer
13       ...                         // deposita o item no buffer
14       signal (item) ;             // sinaliza um novo item
15       unlock (mbuf) ;             // libera o buffer
16    }
17 }
18
19 task consumidor ()
20 {
21    while (1)
22    {
23       lock (mbuf) ;               // obtem o mutex do buffer
24       while (num_items == 0)      // enquanto o buffer estiver vazio
25          wait (item, mbuf) ;      // espera um item, liberando o buffer
26       ...                         // retira o item no buffer
27       signal (vaga) ;             // sinaliza uma vaga livre
28       unlock (mbuf) ;             // libera o buffer
29       ...                         // consome o item retirado do buffer
30    }
31 }
```

JANTAR DOS FILÓSOFOS

- Um dos problemas clássicos de coordenação mais conhecidos é o jantar dos filósofos, proposto inicialmente por Dijkstra [Raynal, 1986; Ben-Ari, 1990].
  - Neste problema, um grupo de cinco filósofos chineses alterna sua vida entre meditar e comer.
  - Há uma mesa redonda com um lugar fixo para cada filósofo, com um prato, cinco palitos (hashis) compartilhados e um grande prato de arroz ao centro.
  - Para comer, um filósofo **fi** precisa pegar o palito à sua direita (**pi**) e à sua esquerda (**pi+1**), um de cada vez.
  - Como os palitos são compartilhados, dois filósofos vizinhos não podem comer ao mesmo tempo. Os filósofos não conversam entre si, nem podem observar os estados uns dos outros.
- O problema do jantar dos filósofos é representativo de uma grande classe de problemas de sincronização entre vários processos e vários recursos sem usar um coordenador central.
- Resolver o problema do jantar dos filósofos consiste em encontrar uma forma de coordenar suas ações de maneira que todos os filósofos consigam meditar e comer.

JANTAR DOS FILÓSOFOS

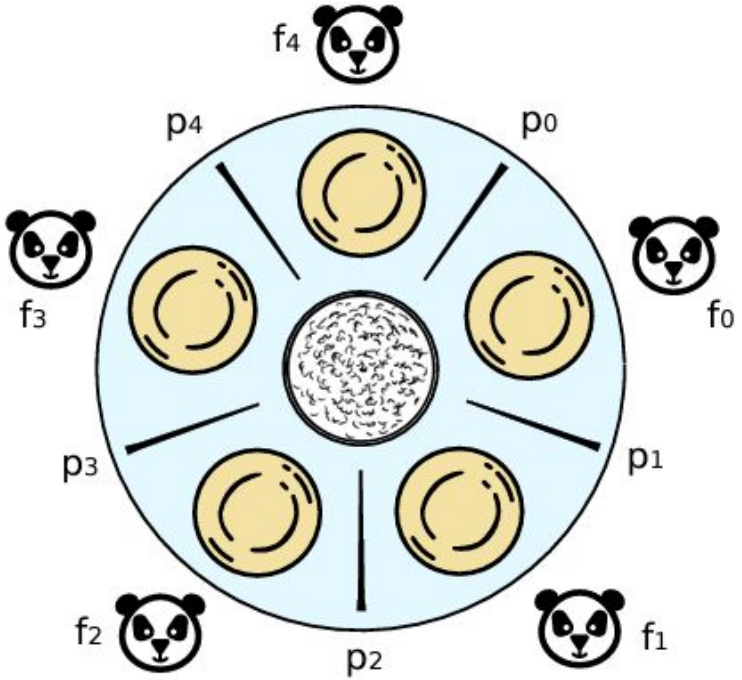


Figura 12.3: O jantar dos filósofos chineses.

JANTAR DOS FILÓSOFOS

```
1  #define NUMFILO 5
2  semaphore hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)
3
4  task filosofo (int i)              // filósofo i (entre 0 e 4)
5  {
6     int dir = i ;
7     int esq = (i+1) % NUMFILO ;
8
9     while (1)
10    {
11       meditar () ;
12       down (hashi [dir]) ;         // pega palito direito
13       down (hashi [esq]) ;         // pega palito esquerdo
14       comer () ;
15       up (hashi [dir]) ;           // devolve palito direito
16       up (hashi [esq]) ;           // devolve palito esquerdo
17    }
18 }
```

JANTAR DOS FILÓSOFOS

- Soluções simples para esse problema podem provocar impasses, ou seja, situações em que todos os filósofos ficam bloqueados.
- Outras soluções podem provocar inanição (starvation), ou seja, alguns dos filósofos nunca conseguem comer.
- A próxima figura apresenta os filósofos em uma situação de impasse: cada filósofo obteve o palito à sua direita e está esperando o palito à sua esquerda (indicado pelas setas tracejadas). Como todos os filósofos estão esperando, ninguém consegue executar suas ações.
- Uma solução trivial para o problema do jantar dos filósofos consiste em colocar um "saleiro" hipotético sobre a mesa: quando um filósofo deseja comer, ele deve obter o saleiro antes de pegar os palitos; assim que tiver ambos os palitos, ele devolve o saleiro à mesa e pode comer.

JANTAR DOS FILÓSOFOS

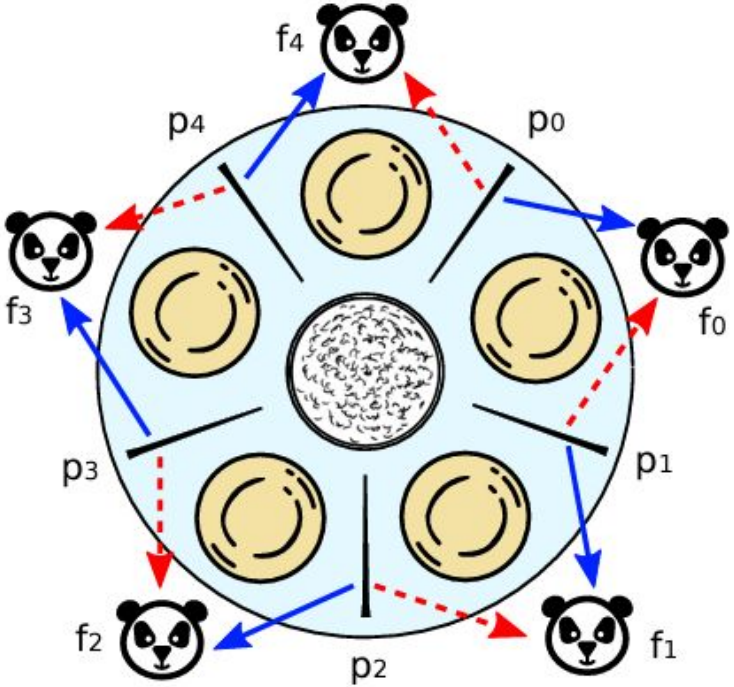


Figura 12.4: Um impasse no jantar dos filósofos chineses.

JANTAR DOS FILÓSOFOS

```
1  #define NUMFILO 5
2  semaphore hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)
3  semaphore saleiro ;         // um semáforo para o saleiro
4
5  task filosofo (int i)              // filósofo i (entre 0 e 4)
6  {
7     int dir = i ;
8     int esq = (i+1) % NUMFILO] ;
9
10    while (1)
11    {
12       meditar () ;
13       down (saleiro) ;             // pega saleiro
14       down (hashi [dir]) ;         // pega palito direito
15       down (hashi [esq]) ;         // pega palito esquerdo
16       up (saleiro) ;               // devolve saleiro
17       comer () ;
18       up (hashi [dir]) ;           // devolve palito direito
19       up (hashi [esq]) ;           // devolve palito esquerdo
20    }
21 }
```

Obviamente, a solução do saleiro serializa o acesso aos palitos e por isso tem baixo desempenho se houverem muitos filósofos disputando o mesmo saleiro. Diversas soluções eficientes podem ser encontradas na literatura para esse problema [Tanenbaum, 2003; Silberschatz et al., 2001].

BARBEIRO DORMINHOCO

- O problema consiste em simular o funcionamento de uma barbearia com as seguintes características:
  - A barbearia tem uma sala de espera com N cadeiras e uma cadeira de barbear.
  - Se não tem clientes à espera, o barbeiro senta numa cadeira e dorme.
  - Quando chega um cliente, ele acorda o barbeiro.
  - Se chega outro cliente enquanto o barbeiro está trabalhando, ele ocupa uma cadeira e espera (se tem alguma cadeira disponível) ou vai embora (se todas as cadeiras estão ocupadas).

BARBEIRO DORMINHOCO

- A solução a seguir usa 3 semáforos: **clientes**, **fila** e **mutex**.
  - O semáforo **clientes** tranca o barbeiro, sendo suas “bolitas” produzidas pelos clientes que chegam.
    - O valor desse semáforo indica o número de clientes à espera (excluindo o cliente na cadeira do barbeiro, que não está à espera).
  - O semáforo **fila** tranca os clientes e implementa a fila de espera.
  - O semáforo **mutex** garante exclusão mútua.
  - Também é usada uma variável inteira, count, que conta o número de clientes à espera. O valor desta variável é sempre igual ao “número de bolitas” do semáforo **clientes**.

BARBEIRO DORMINHOCO

- Visão geral do algoritmo:
  - Um cliente que chega na barbearia verifica o número de clientes à espera.
  - Se esse número é menor que o número de cadeiras, o cliente espera, caso contrário, ele vai embora.
    - A solução é apresentada a seguir, considerando o número de cadeiras na sala de espera igual a 3.
  - Inicialmente, o barbeiro executa a operação P(clientes), onde fica bloqueado (dormindo) até a chegada de algum cliente.
  - Quando chega um cliente, ele começa adquirindo a exclusão mútua.
  - Outro cliente que chegar imediatamente após, irá se bloquear até que o primeiro libere a exclusão mútua.
  - Dentro da região crítica, o cliente verifica se o número de pessoas à espera é menor ou igual ao número de cadeiras.
  - Se não é, ele libera mutex e vai embora sem cortar o cabelo.

BARBEIRO DORMINHOCO

- Visão geral do algoritmo (continuação):
  - Se tem alguma cadeira disponível, o cliente incrementa a variável count e executa a operação V no semáforo clientes.
  - Se o barbeiro está dormindo, ele é acordado; caso contrário, é adicionada uma “bolita” no semáforo clientes.
  - A seguir, o cliente libera a exclusão mútua e entra na fila de espera.
  - O barbeiro adquire a exclusão mútua, decrementa o número de clientes, pega o primeiro da fila de espera e vai fazer o corte.
  - Quando termina o corte de cabelo, o cliente deixa a barbearia e o barbeiro repete o seu loop onde tenta pegar um próximo cliente. Se tem cliente, o barbeiro faz outro corte. Se não tem, o barbeiro dorme.

BARBEIRO DORMINHOCO

Variáveis globais: *clientes*, *fila*: semaphore init 0;
mutex: semaphore init 1;
count : integer initial 0;

Processo barbeiro:

```
loop
   P(clientes);   /*dorme, se for o caso*/
   P(mutex);
      count:= count –1;
      V(fila);   /*pega próximo cliente*/
   V(mutex);
   /*corta o cabelo*/
endloop
```

Processo cliente:

```
P(mutex);
if count < 3
then  { count:= count+1;
        V(clientes);   /*acorda o barbeiro*/
        V(mutex);
        P(fila);       /*espera o barbeiro*/
        /*corta o cabelo*/
      }
else  V(mutex)
```

LEITORES E ESCRITORES

- Outra situação que ocorre com frequência em sistemas concorrentes é o problema dos leitores/escritores. [
- Neste problema, um conjunto de tarefas acessa de forma concorrente uma área de memória compartilhada, na qual podem fazer leituras ou escritas de valores.
- De acordo com as condições de Bernstein, as leituras podem ser feitas em paralelo, pois não interferem umas com as outras, mas as escritas devem ser realizadas com acesso exclusivo à área compartilhada, para evitar condições de disputa.
- A próxima figura mostra leitores e escritores acessando de forma concorrente uma matriz de números inteiros M.
- O estilo de sincronização leitores/escritores é encontrado com muita frequência em aplicações com múltiplas threads.
- O padrão POSIX define mecanismos para a criação e uso de travas com essa funcionalidade, acessíveis através de chamadas como `pthread_rwlock_init()`, entre outras.

LEITORES E ESCRITORES

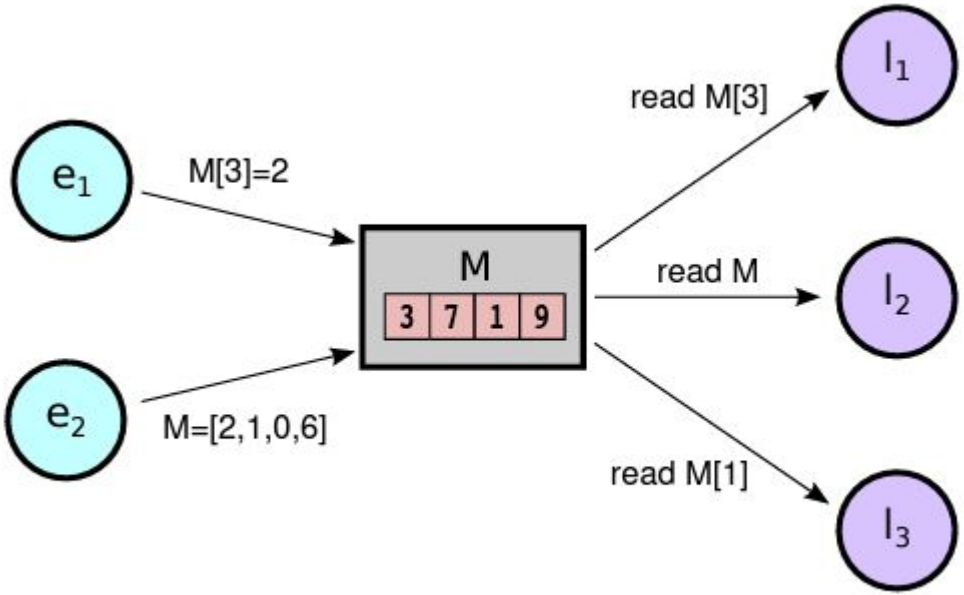


Figura 12.2: O problema dos leitores/escritores.

LEITORES E ESCRITORES

- Solução simplista:
  - Uma solução simplista para esse problema consistiria em proteger o acesso à área compartilhada com um mutex ou semáforo inicializado em 1; assim, somente um leitor ou escritor poderia acessar a área de cada vez.
  - Essa solução deixa muito a desejar em termos de desempenho, porque restringe desnecessariamente o acesso dos leitores à área compartilhada: como a operação de leitura não altera os valores armazenados, não haveria problema em permitir o acesso paralelo de vários leitores à área compartilhada, desde que as escritas continuem sendo feitas de forma exclusiva.

LEITORES E ESCRITORES

```
1  mutex marea ;                // controla o acesso à área
2
3  task leitor ()
4  {
5     while (1)
6     {
7        lock (marea) ;         // requer acesso exclusivo à área
8        ...                    // lê dados da área compartilhada
9        unlock (marea) ;       // libera o acesso à área
10       ...
11    }
12 }
13
14 task escritor ()
15 {
16    while (1)
17    {
18       lock (marea) ;         // requer acesso exclusivo à área
19       ...                    // escreve dados na área compartilhada
20       unlock (marea) ;       // libera o acesso à área
21       ...
22    }
23 }
```

LEITORES E ESCRITORES

- Solução com priorização de leitores:
  - Uma solução melhor para o problema dos leitores/escritores, considerando a possibilidade de acesso paralelo pelos leitores, seria a indicada na listagem a seguir.
  - Nela, os leitores dividem a responsabilidade pelo mutex de controle da área compartilhada (marea):
    - o primeiro leitor a entrar obtém esse mutex, que só será liberado pelo último leitor a sair da área.
    - Um contador de leitores permite saber se um leitor é o primeiro a entrar ou o último a sair.
    - Como esse contador pode sofrer condições de disputa, o acesso a ele é controlado por outro mutex (mcont).
  - Essa solução melhora o desempenho das operações de leitura, pois permite que vários leitores acessem simultaneamente.
  - Contudo, introduz um novo problema: a priorização dos leitores. De fato, sempre que algum leitor estiver acessando a área compartilhada, outros leitores também podem acessá-la, enquanto eventuais escritores têm de esperar até que a área fique livre (sem leitores).
  - Caso existam muitos leitores em atividade, os escritores podem ficar impedidos de acessar a área, pois ela nunca ficará vazia (inanição).

LEITORES E ESCRITORES

```
1  mutex marea ;              // controla o acesso à área
2  mutex mcont ;              // controla o acesso ao contador
3
4  int num_leitores = 0 ;     // número de leitores acessando a área
5
6  task leitor ()
7  {
8     while (1)
9     {
10       lock (mcont) ;        // requer acesso exclusivo ao contador
11       num_leitores++ ;      // incrementa contador de leitores
12       if (num_leitores == 1) // sou o primeiro leitor a entrar?
13          lock (marea) ;     // requer acesso à área
14       unlock (mcont) ;      // libera o contador
15
16       ...                   // lê dados da área compartilhada
17
18       lock (mcont) ;        // requer acesso exclusivo ao contador
19       num_leitores-- ;      // decrementa contador de leitores
20       if (num_leitores == 0) // sou o último leitor a sair?
21          unlock (marea) ;   // libera o acesso à área
22       unlock (mcont) ;      // libera o contador
23       ...
24    }
25 }
26
27 task escritor ()
28 {
29    while (1)
30    {
31       lock (marea) ;        // requer acesso exclusivo à área
32       ...                   // escreve dados na área compartilhada
33       unlock (marea) ;      // libera o acesso à área
34       ...
35    }
36 }
```