Capítulo 12

Problemas clássicos

Algumas situações de coordenação entre tarefas ocorrem com muita frequência na programação de sistemas complexos. Os *problemas clássicos de coordenação* retratam muitas dessas situações e permitem compreender como podem ser implementadas suas soluções.

Este capítulo apresenta alguns problemas clássicos: o problema dos *produtores/-consumidores*, o problema dos *leitores/escritores* e o *jantar dos filósofos*. Diversos outros problemas clássicos são frequentemente descritos na literatura, como o *problema dos fumantes* e o do *barbeiro dorminhoco*, entre outros [Raynal, 1986; Ben-Ari, 1990]. Uma extensa coletânea de problemas de coordenação e suas soluções é apresentada em [Downey, 2016], disponível *online*.

12.1 Produtores/consumidores

Este problema também é conhecido como o problema do *buffer limitado*, e 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 o mesmo tenha uma vaga livre. Caso contrário, deve esperar até que surja uma vaga. Ao depositar um item, o produtor "consome" uma vaga livre.

Consumidor: retira um item do *buffer* e o consome; caso o *buffer* esteja 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 fazer seu acesso, seja para produzir ou para consumir um item. A Figura 12.1 ilustra esse problema, envolvendo vários produtores e consumidores acessando um *buffer* com capacidade para 9 ítens. É interessante observar a forte similaridade dessa figura com o *Mailbox* da Figura 8.10; na prática, a implementação de *mailboxes* e de *pipes* é geralmente feita usando um esquema de sincronização produtor/consumidor.

A solução do problema dos produtores/consumidores envolve três aspectos de coordenação distintos e complementares:

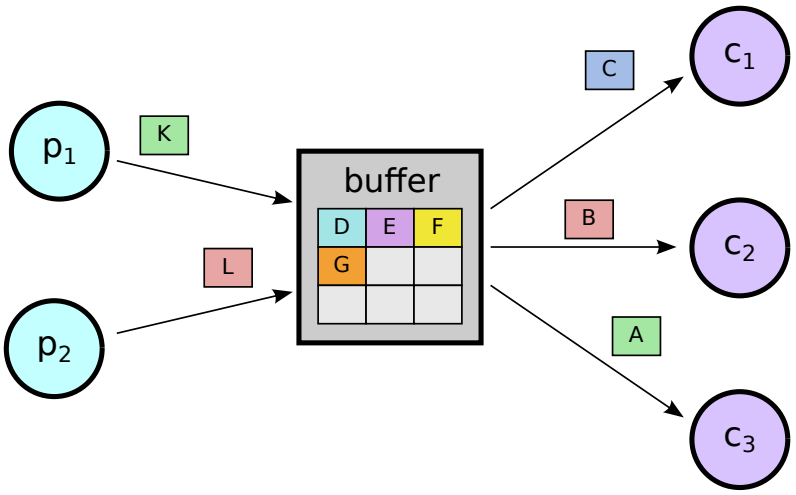

Figura 12.1: O problema dos produtores/consumidores.

- 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 do *buffer* estar cheio: os produtores devem esperar até que surjam vagas livres no *buffer*.
- A suspensão dos consumidores no caso do *buffer* estar vazio: os consumidores devem esperar até que surjam novos itens a consumir no *buffer*.

12.1.1 Solução usando semáforos

Pode-se resolver o problema dos produtores/consumidores de forma eficiente usando um *mutex* e dois semáforos, um para cada aspecto de coordenação envolvido. O código a seguir ilustra de forma simplificada uma solução para esse problema, considerando um buffer com capacidade para N itens, inicialmente vazio:

```
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 }
```

É importante observar que essa solução é genérica, pois não depende do tamanho do buffer, do número de produtores nem do número de consumidores.

12.1.2 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.

```
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 }
```

12.2 Leitores/escritores

Outra situação que ocorre com frequência em sistemas concorrentes é o problema dos *leitores/escritores*. Neste problema, um conjunto de tarefas acessam de forma concorrente uma área de memória compartilhada, na qual podem fazer leituras ou escritas de valores. De acordo com as condições de Bernstein (Seção 10.1.3), as leituras podem ser feitas em paralelo, pois não interferem umas com as outras, mas as escritas têm de ser feitas com acesso exclusivo à área compartilhada, para evitar condições de disputa. A Figura 12.2 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.

12.2.1 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

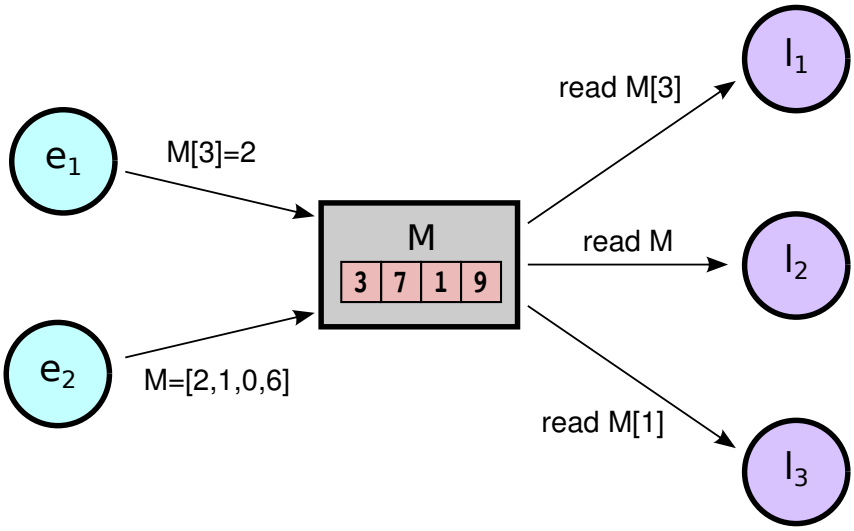

Figura 12.2: O problema dos leitores/escritores.

processo por vez poderia acessar a área, garantindo a integridade de todas as operações. O código a seguir ilustra essa abordagem:

```
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 }
```

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.

12.2.2 Solução com priorização dos 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`).

```
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 }
```

Essa solução melhora o desempenho das operações de leitura, pois permite que vários leitores a 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 a área ficar livre (sem leitores). Caso existam muito leitores em atividade, os escritores podem ficar impedidos de acessar a área, pois ela nunca ficará vazia (inanição).

Soluções com priorização para os escritores e soluções equitativas entre ambos podem ser facilmente encontradas na literatura [Raynal, 1986; Ben-Ari, 1990].

12.3 O jantar dos selvagens

Uma variação curiosa do problema dos produtores/consumidores foi proposta em [Andrews, 1991] com o nome de *Jantar dos Selvagens*: uma tribo de selvagens está jantando ao redor de um grande caldeirão contendo N porções de missionário cozido. Quando um selvagem quer comer, ele se serve de uma porção no caldeirão, a menos que este esteja vazio. Nesse caso, o selvagem primeiro acorda o cozinheiro da tribo e espera que ele encha o caldeirão de volta, para então se servir novamente. Após encher o caldeirão, o cozinheiro volta a dormir.

```
1  task cozinheiro ()
2  {
3     while (1)
4     {
5        encher_caldeirao () ;
6        dormir () ;
7     }
8  }
9
10 task selvagem ()
11 {
12    while (1)
13    {
14       servir () ;
15       comer () ;
16    }
17 }
```

As restrições de sincronização deste problema são as seguintes:

- Selvagens não podem se servir ao mesmo tempo (mas podem comer ao mesmo tempo);
- Selvagens não podem se servir se o caldeirão estiver vazio;
- O cozinheiro só pode encher o caldeirão quando ele estiver vazio.

Uma solução simples para esse problema, apresentada em [Downey, 2016], é a seguinte:

```
1  int porcoes = 0 ;           // porções no caldeirão
2  mutex mcald ;               // controla acesso ao caldeirão
3  semaphore cald_vazio ;      // indica caldeirão vazio (inicia em 0)
4  semaphore cald_cheio ;      // indica caldeirão cheio (inicia em 0)
5
6  task cozinheiro ()
7  {
8     while (1)
9     {
10       down (cald_vazio) ;   // aguarda o caldeirão esvaziar
11       porcoes += M ;        // enche o caldeirão (M porções)
12       up (cald_cheio) ;     // avisa que encheu o caldeirão
13    }
14 }
15
16
17 task selvagem ()
18 {
19    while (1)
20    {
21       lock (mcald) ;        // tenta acessar o caldeirão
22       if (porcoes == 0)     // caldeirão vazio?
23       {
24          up (cald_vazio) ;  // avisa que caldeirão esvaziou
25          down (cald_cheio) ; // espera ficar cheio de novo
26       }
27       porcoes-- ;           // serve uma porção
28       unlock (mcald) ;      // libera o caldeirão
29       comer () ;
30    }
31 }
```

12.4 O 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 suas vidas entre meditar e comer.

Ha uma mesa redonda com um lugar fixo para cada filósofo, com um prato, cinco palitos (*hashis*) compartilhados e um grande prato de arroz ao meio[1]. Para comer, um filósofo f_i precisa pegar o palito à sua direita (p_i) e à sua esquerda (p_{i+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. A Figura 12.3 ilustra essa situação.

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. A listagem a seguir é o pseudocódigo de uma implementação do comportamento básico dos filósofos, na qual cada filósofo é uma tarefa e palito é um semáforo:

[1] Na versão inicial de Dijkstra, os filósofos compartilhavam garfos e comiam spaghetti; neste texto os filósofos são chineses e comem arroz...

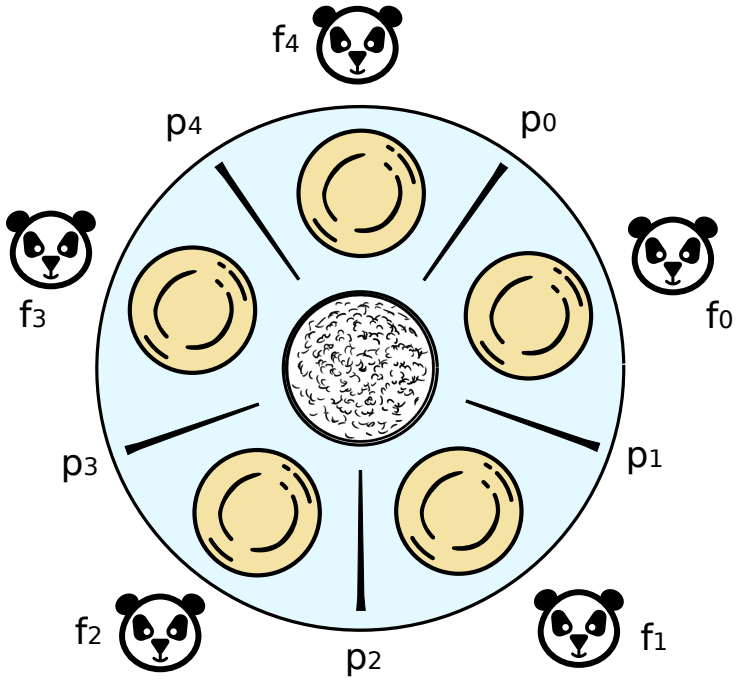

Figura 12.3: O jantar dos filósofos chineses.

```
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 }
```

Soluções simples para esse problema podem provocar impasses, ou seja, situações nas quais todos os filósofos ficam bloqueados (impasses serão estudados na Seção 13). Outras soluções podem provocar inanição (*starvation*), ou seja, alguns dos filósofos nunca conseguem comer. A Figura 12.4 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 mais consegue executar.

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 obter os palitos; assim que tiver ambos os palitos, ele devolve o saleiro à mesa e pode comer:

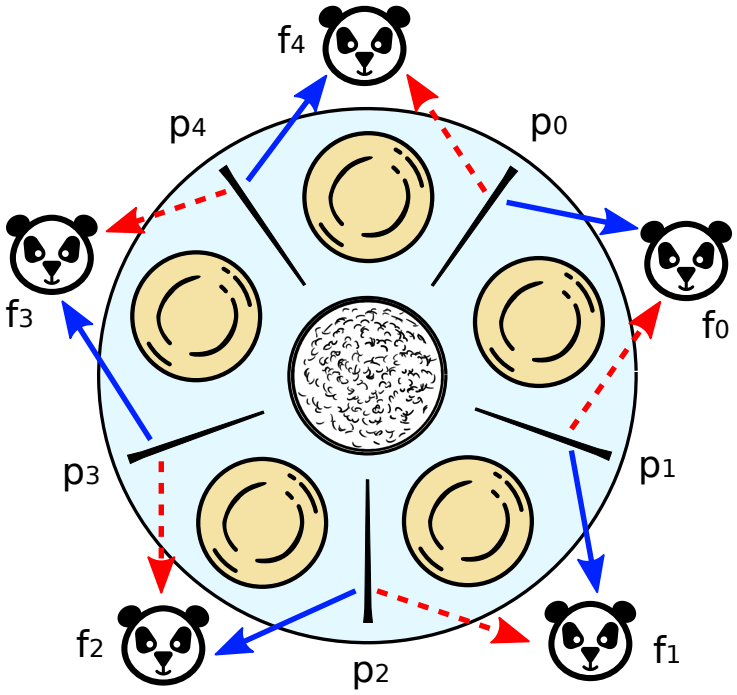

Figura 12.4: Um impasse no jantar dos filósofos chineses.

```
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].

Exercícios

1. Usando semáforos, escreva o pseudo-código de um sistema produtor/consumidor com dois buffers limitados organizado na forma $X \rightarrow B_1 \rightarrow Y \rightarrow B_2 \rightarrow Z$, onde X, Y e Z são tipos de processos e B_1 e B_2 são buffers independentes com capacidades N_1 e N_2, respectivamente, inicialmente vazios. Os buffers são acessados unicamente através das operações *insere(B_i, item)* e *retira(B_i, item)* (que não precisam ser detalhadas). O número de processos X, Y e Z é desconhecido.

   Devem ser definidos os códigos dos processos X, Y e Z e os semáforos necessários, com seus significados e valores iniciais.

2. O trecho de código a seguir apresenta uma solução para o problema do jantar dos filósofos, mas essa solução apresenta risco de impasse. Explique por que e como podem ocorrer impasses nessa solução. Em seguida, modifique o código para eliminar esse risco.

```
1  #define N 5
2
3  sem_t garfo[5] ; // 5 semáforos iniciados em 1
4
5  void filosofo (int i)
6  {
7     while (1)
8     {
9        medita ();
10       sem_down (garfo [i]) ;
11       sem_down (garfo [(i+1) % N]) ;
12       come ();
13       sem_up (garfo [i]) ;
14       sem_up (garfo [(i+1) % N]) ;
15    }
16 }
```

3. Suponha três robôs (*Bart*, *Lisa*, *Maggie*), cada um controlado por sua própria *thread*. Você deve escrever o código das *threads* de controle, usando semáforos para garantir que os robôs se movam sempre na sequência *Bart* $\rightarrow$ *Lisa* $\rightarrow$ *Maggie* $\rightarrow$ *Lisa* $\rightarrow$ *Bart* $\rightarrow$ *Lisa* $\rightarrow$ *Maggie* $\rightarrow \cdots$, um robô de cada vez. Use a chamada `move()` para indicar um movimento do robô. Não esqueça de definir os valores iniciais das variáveis e/ou dos semáforos utilizados. Soluções envolvendo espera ocupada (*busy wait*) não devem ser usadas.

4. O *Rendez-Vous* é um operador de sincronização forte entre **dois** processos ou *threads*, no qual um deles espera até que ambos cheguem ao ponto de encontro (*rendez-vous*, em francês). O exemplo a seguir ilustra seu uso:

```
Processo A                    Processo B
   A1 () ;                       B1 () ;
   rv_wait (rv) ;                rv_wait (rv) ;
   A2 () ;                       B2 () ;
   rv_wait (rv) ;                rv_wait (rv) ;
   A3 () ;                       B3 () ;
```

Considerando a relação $a \rightarrow b$ como "a ocorre antes de b" e a relação $a \parallel b$ como "a e b ocorrem sem uma ordem definida", temos as seguintes restrições de sincronização:

- $\forall(i,j), A_i \rightarrow B_{j>i}$ e $B_i \rightarrow A_{j>i}$ (imposto pelo *Rendez-Vous*)
- $\forall(i,j), A_i \rightarrow A_{j>i}$ e $B_i \rightarrow B_{j>i}$ (imposto pela execução sequencial)
- $\forall(i,j), A_i \parallel B_{j=i}$ (possibilidade de execução concorrente)

Escreva o pseudo-código necessário para implementar *Rendez-Vous*, usando semáforos ou mutexes. Não esqueça de inicializar as variáveis e semáforos utilizados. Soluções que incorram em espera ocupada (*busy wait*) não devem ser usadas.

```
1  // estrutura que representa um RV
2  typedef struct rv_t
3  {
4     ... // completar
5  } rv_t ;
6
7  // operador de espera no RV
8  void rv_wait (rv_t *rv)
9  {
10    ... // completar
11 }
12
13 // inicialização do RV
14 void rv_init (rv_t *rv)
15 {
16    ... // completar
17 }
```

5. Uma *Barreira* é um operador de sincronização forte entre **N** processos ou *threads*, no qual eles esperam até que todos cheguem à barreira. O exemplo a seguir ilustra seu uso:

```
Processo A                    Processo C
   A1 () ;                       C1 () ;
   barrier_wait (b) ;            barrier_wait (b) ;
   A2 () ;                       C2 () ;
   barrier_wait (b) ;            barrier_wait (b) ;
   A3 () ;                       C3 () ;

Processo B                    Processo D
   B1 () ;                       D1 () ;
   barrier_wait (b) ;            barrier_wait (b) ;
   B2 () ;                       D2 () ;
   barrier_wait (b) ;            barrier_wait (b) ;
   B3 () ;                       D3 () ;
```

Considerando a relação $a \rightarrow b$ como "a ocorre antes de b" e a relação $a \parallel b$ como "a e b ocorrem sem uma ordem definida", temos as seguintes restrições de sincronização:

- $\forall(i, j), X \neq Y, X_i \rightarrow Y_{j>i}$ (imposto pela barreira)
- $\forall(i, j), X_i \rightarrow X_{j>i}$ (imposto pela execução sequencial)
- $\forall(i, j), X \neq Y, X_i \parallel Y_{j=i}$ (possibilidade de execução concorrente)

Escreva o pseudo-código necessário para implementar barreiras para N processos, usando semáforos ou mutexes. Não esqueça de inicializar as variáveis e semáforos utilizados. Soluções que incorram em espera ocupada (*busy wait*) não devem ser usadas.

```
1  // estrutura que representa uma barreira
2  typedef struct barrier_t
3  {
4     ... // completar
5  } barrier_t ;
6
7  // operador de espera na barreira
8  void barrier_wait (barrier_t *barrier)
9  {
10    ... // completar
11 }
12
13 // inicialização de barreira para N processos
14 void barrier_init (barrier_t *barrier, int N)
15 {
16    ... // completar
17 }
```

Atividades

1. Implemente uma solução em C para o problema do produtor/consumidor, usando threads e semáforos no padrão POSIX.
2. Implemente uma solução em C para o problema do produtor/consumidor, usando threads e variáveis de condição no padrão POSIX.
3. Implemente uma solução em C para o problema dos leitores/escritores com priorização para escritores, usando threads e semáforos POSIX.
4. Implemente uma solução em C para o problema dos leitores/escritores com priorização para escritores, usando threads e *rwlocks* POSIX.

Referências

G. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.

M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.

A. Downey. *The Little Book of Semaphores*. Green Tea Press, 2016. urlhttp://greenteapress.com/wp/semaphores/.

M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

A. Silberschatz, P. Galvin, and G. Gagne. *Sistemas Operacionais – Conceitos e Aplicações*. Campus, 2001.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.