AULA 09

Sincronização de Processos - II

Monitores

Conforme comentamos, o uso equivocado dos semáforos pode levar a uma situação de *deadlock*, por isso devemos tomar cuidado ao programar utilizando este mecanismo de sincronização para evitar situações desagradáveis e, muitas vezes, difícies de serem analisadas.

Em meados da década de 70, Hoare e Brinch Hansen propuseram uma primitiva de alto nível para coordenar a sincronização de processos e facilitar a escrita de programas paralelos. Tais primitivas foram denominadas de *monitores*. A idéia das propostas de cada um são bem semelhantes.

Um monitor é um conjunto de procedimentos, variáveis e estrutura de dados, todas agrupadas em um módulo especial. Os processos podem chamar os procedimentos do monitor sempre que desejarem, mas nunca poderão acessar diretamente as estruturas de dados e as variáveis internas do monitor a partir de procedimentos declarados fora do monitor [1].

Relacionado à sincronização dos processos e exclusão mútua, os monitores oferecem uma propriedade importante: apenas um processo pode estar ativo dentro do monitor em um determinado instante. Toda as verificações são efetuadas pelo compilador e pelo fato do compilador implementar a exclusão mútua é muito pouco provável que algo venha a dar errado, como pode acontecer no uso dos semáforos. Portanto, para o programador basta colocar a região crítica dentro do monitor que não haverá a possibilidade de existir 2 ou mais processos executando a região crítica ao mesmo tempo.

Funcionamento: quando um processo realiza uma chamada a algum procedimento do monitor este verifica se existe se algum outro processo está ativo dentro dele. Caso exista, o processo que está tentando entrar no monitor é suspenso até que o outro processo libere este monitor. Se não existir nenhum processo ativo dentro do monitor, o processo que está solicitando o acesso poderá entrar e executar o procedimento chamado.

Apesar dos monitores também garantirem a exclusão mútua é necessário cuidarmos dos casos onde devemos bloquear os processos quando não houver mais condições para que os mesmos continuem executando (ex. Buffer vazio ou cheio). Para resolver este problema propuseram o

uso de *variáveis de condição*, com as seguintes operações sobre elas: WAIT e SIGNAL. As semânticas dessas operações são: bloqueio (semelhante ao *sleep*) e desbloqueio (semelhante ao *wakeup*) com uso de filas.

Brinch Hansen propôs que ao executar a operação SIGNAL o processo seria obrigado a deixar o monitor imediatamente em seguida. Se um SIGNAL é executado sobre uma variável de condição, somente um processo associado a fila de espera desta variável poderá ser reativado pelo escalonador de processos.

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;

        count := 0;
end monitor;

procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;

procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Produtor consumidor com monitores.

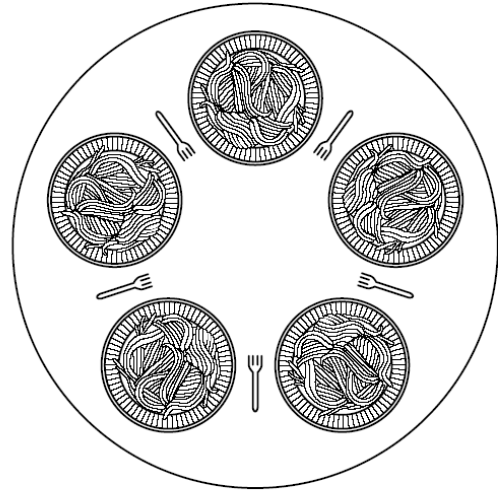
Problema: poucas linguagens dão suporte ao uso de monitores, Java é uma que usa monitores para realizar sincronização entre threads. A linguagem C não possui monitores.

```
public class ProducerConsumer {
        static final int N = 100;      // constant giving the buffer size
        static producer p = new producer( );    // instantiate a new producer thread
        static consumer c = new consumer( ); // instantiate a new consumer thread
        static our_monitor mon = new our_monitor( );      // instantiate a new monitor

        public static void main(String args[ ]) {
           p.start( );      // start the producer thread
           c.start( );      // start the consumer thread
        }

        static class producer extends Thread {
           public void run( )  {// run method contains the thread code
              int item;
              while (true) {      // producer loop
                 item = produce_item( );
                 mon.insert(item);
              }
           }
           private int produce_item( ) { ... }      // actually produce
        }

        static class consumer extends Thread {
           public void run( )  {run method contains the thread code
              int item;
              while (true) {      // consumer loop
                 item = mon.remove( );
                 consume_item (item);
              }
           }
           private void consume_item(int item) { ... }        // actually consume
        }

        static class our_monitor {  // this is a monitor
           private int buffer[ ] = new int[N];
           private int count = 0, lo = 0, hi = 0;  // counters and indices

           public synchronized void insert(int val) {
              if (count == N) go_to_sleep( );    // if the buffer is full, go to sleep
              buffer [hi] = val; // insert an item into the buffer
              hi = (hi + 1) % N;       // slot to place next item in
              count = count + 1;      // one more item in the buffer now
              if (count == 1) notify( );       // if consumer was sleeping, wake it up
           }
```

Produtor Consumidor em Java

Problemas Clássicos de Comunicação entre Processos

1. O problema dos Filósofos Glutões / Jantar dos Filósofos



Considere cinco filósofos que passam suas vidas meditando e comendo. Cada um dos filósofos compartilham uma mesa redonda comum cercada por cinco cadeiras. Cada filosofo tem um prato de espaguete. Para que o filósofo consiga comer ele precisa de 2 garfos. A vida de cada filósofo consiste em períodos nos quais ele alternadamente come e pensa. Quando ele medita ele não interage com seus colegas. Quando um filósofo sente fome ele tenta pegar os dois garfos mais próximos de si (direita e esquerda do seu próprio prato). Um filósofo pode pegar apenas um garfo de cada vez (a ordem não importa) e, é claro, não poderá pegar nenhum garfo que esteja na mão de outro filósofo. Assim, quando um filósofo está de posse de dois garfos ele poderá comer. Passado um tempo ele pára de comer, solta os garfos e volta a pensar [1].

Este exemplo pode ser aplicado a uma vasta classe de problemas de controle de concorrência. Como podemos verificar é uma representação simples do problema de alocar vários recursos a vários processos, sem causar deadlocks.

Quais soluções podem ser possíveis para este problema?

1. pegar garfo da esquerda e depois o da direita => se todos resolvem pegar o garfo da esquerda ao mesmo tempo gera um deadlock.

2. pegar o garfo da esquerda e tentar pegar o da direita, se estiver sendo usado o filósofo devolve o garfo e espera um tempo para tentar mais tarde. Se todos estão sincronizados e passam a esperar o mesmo tempo para tentar novamente o impasse continuará.

3. ao invés de usar o mesmo tempo de espera uma estratégia que funciona é esperar um tempo aleatório para tentar novamente. Usar semáforos para controlar o acesso aos garfos.

Aplicação: processos que competem por um número limitado de recursos (dispositivos de E/S, por exemplo).

O problema dos Leitores e Escritores

Este problema é muito interessante para simular o acesso a uma grande base de dados, tal como um sistema de reserva de passagens, onde temos muitos processos competindo para ler e escrever nela. Intuitivamente, podemos presumir que não existe nenhum problema se dois ou mais processos estiverem lendo a base de dados ao mesmo tempo, mas se algum processo estiver escrevendo (escritor) algo na base de dados, nenhum outro processo, inclusive os leitores, poderão ter acesso enquanto o escritor não terminar. Tudo isso deve ser respeitado para manter a consistência das informações do sistema.

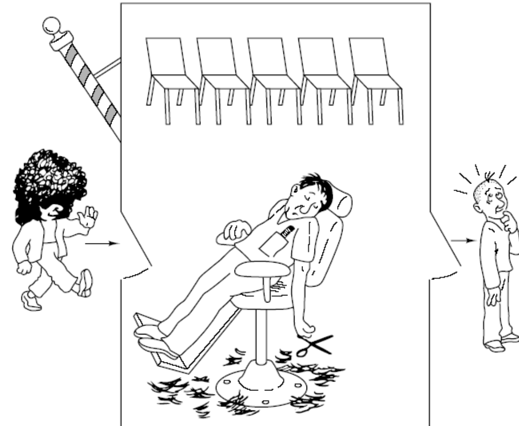
Como resolver o problema? Veja a solução dada a seguir [1]:

```
typedef int semaphore;          /* use your imagination */
semaphore mutex = 1;            /* controls access to 'rc' */
semaphore db = 1;               /* controls access to the database */
int rc = 0;                     /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {              /* repeat forever */
        down(&mutex);           /* get exclusive access to 'rc' */
        rc = rc + 1;            /* 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 = rc – 1;            /* one reader fewer now */
        if (rc == 0) up(&db);   /* if this is the last reader ... */
        up(&mutex);             /* release exclusive access to 'rc' */
        use_data_read( );       /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {              /* repeat forever */
        think_up_data( );       /* noncritical region */
        down(&db);              /* get exclusive access */
        write_data_base( );     /* update the data */
        up(&db);                /* release exclusive access */
    }
}
```

Esta solução dá prioridade ao leitores, inibindo os escritores de usar a RC enquanto existir algum leitor ativo.

O problema do Barbeiro Dorminhoco

Existe uma barbearia com uma cadeira de barbeiros e diversos lugares para que os clientes possam esperar, se houver algum, sentados. Caso não tenha clientes, o barbeiro senta em sua cadeira e dorme. Quando chegarem fregueses, enquanto o barbeiro estiver cortando o cabelo de outro, estes devem ou sentar, se houver cadeira vazia, ou ir embora, se não houver nenhuma cadeira livre. Como programar uma solução para o problema? Veja a solução abaixo:



```
#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 */
    }
}
```

Referência Bibliográfica

[1] TANENBAUM, A. S. **Sistemas Operacionais Modernos**. 2ª ed. São Paulo: Prentice Hall, 2003.

[2] SILBERSCHATZ, A. et al. **Sistemas Operacionais: conceitos e aplicações**. Campus, 2001.