AULA 08 - Sincronização de Processos

Semáforos

Em 1965 Dijkstra propôs usar uma variável inteira para contar o número de sinais de acordar e salvá-los para uso futuro. Esta variável recebeu o nome de ***semáforo*** e é usada para sincronizar processos. Seu objetivo é coordenar o acesso a uma região crítica e evitar os problemas que levam a condições de corrida. Esta variável pode armazenar o valor 0 (indicando que nenhum sinal de acordar foi salvo) ou outro valor positivo caso 1 ou mais sinais de acordar estejam pendentes. Além disso, Dijkstra propôs as seguintes operações sobre esta variável chamada semáforo:

1. **DOWN**: verifica se o valor do semáforo é maior que 0, se for **decrementa** o valor do semáforo e prossegue. Caso semáforo = 0 o processo irá dormir sem terminar de executar o DOWN (por enquanto). As operações: verificar valor do semáforo e alterar seu conteúdo são tarefas executadas como uma *ação atômica e indivisível*. Com isso, evitamos o problema de uma interrupção entre essas duas atividades.
2. **UP**: esta operação **incrementa** o valor do semáforo em uma unidade e acorda um dos processos que estão dormindo associado ao semáforo em questão. Assim, um processo é escolhido pelo sistema e é dado a permissão de terminar a execução da operação DOWN que ele havia começado anteriormente. Para terminar, o processo acordado agora irá decrementar o valor do semáforo, fazendo com que ele permaneça em 0 (zero). No entanto, existirá um processo a menos dormindo no semáforo.

Assim, o semáforo é uma solução para o problema da perda do sinal de acordar. Além disso, para ser indivisível as operações UP e DOWN são implementadas como chamadas de sistema. Com isso, por um pequeno instante as interrupções são desabilitadas, mas não geram nenhum problema. Os semáforos binários podem ser usados para controlar 2 ou mais processos que disputem o acesso a uma RC.

Os SOs geralmente fazem distinção entre *semáforos de contagem* e *semáforos binários*. O valor de um semáforo binário pode ser apenas 0 ou 1, enquanto que no outro não existe restrição.

Exemplo do uso:

```
Semaphore S = 1;

DOWN(S);
CriticalSection();
UP(S);
```

Produtor-Consumidor Usando Semáforos

Para que se consiga a atomicidade, as operações sobre semáforos são feitas via chamadas de sistema, onde o sistema operacional inibirá as interrupções enquanto o semáforo estiver sendo utilizado (tempo muito pequeno).

```
define N 100                /* número de posições do buffer*/
typedef int semaphore;
semaphore mutex = 1;        /*controla o acesso a RC*/
semaphore empty = N;        /*conta as posições vazias do buffer*/
semaphore full = 0;         /*conta as posições ocupadas do buffer*/

void producer (void)
{
     int item;
     while (TRUE){
          item = produce_item( );
          down(&empty);    /*decrementa o contador*/
          down(&mutex); /*entra na RC*/
          enter_item(item); /*coloca novo item no buffer*/
          up(&mutex);      /*deixa a RC*/
          up(&full);       /*incrementa    o    contador    de    lugares
                           preenchidos*/
     }
}

void consumer(void)
{
     int item;
     while (TRUE){
          down(&full);     /*decrementa o contador*/
          down(&mutex);    /*entra na RC*/
          item = remove_item( ); /*retira 1 item do buffer*/
          up(&mutex);      /*deixa a RC*/
          up(&empty);      /*incrementa o contador*/
          consume_item(item);
     }
}
```

A cada semáforo existe uma lista associada. Quando um processo é bloqueado ele é colocado nesta lista e o estado do processo é passado para o estado de espera. Depois disso, o controle é transferido para o escalonador que seleciona outro processo para executar.

Um processo bloqueado, que está esperando em um semáforo S, deve ser retomado quando algum outro processo executar UP(S). Com isso, o processo é reiniciado por uma operação *wakeup*, que muda o estado do processo de espera para pronto. Isso elimina o problema da espera ocupada. O uso do semáforo é voltado para **sincronização**. Para garantir exclusão mútua usamos uma versão mais simples dos semáforos denominada mutex.

Mutex (Mutual Exclusion)

Mutex é uma versão simplificada dos semáforos que *serve para controlar o acesso a RC* e que pode estar somente em dois estados distintos: impedido ou desempedido. Portanto, apenas um bit é necessário para armazenar esta informação. Existe 2 procedimentos para usar o esquema de mutex:

1. mutex_clock: se o mutex estiver desempedido (RC livre) o processo ganha acesso a RC. Caso o mutex estiver impedido o processo será bloqueado.
2. mutex_unlock: procedimento chamado pelo processo que deixa a RC para liberar o acesso aos demais processos.

Deadlocks

Podemos definir uma situação de *deadlock* ou impasse como segue [2]:

"*Um conjunto de processos está em uma situação de deadlock, se cada processo do conjunto estiver esperando por um evento que somente outro processo pertencente ao conjunto poderá fazer acontecer"*

Dijkstra define deadlock como: "Efeito colateral de estratégias de sincronização, porque dois processos ao atualizar variáveis compartilhadas, usam uma lock flag, um para requisitar valores de uma variável, o outro para atualizar valor da variável".

Como todos os processos estarão esperando, nenhum deles poderá fazer acontecer qualquer um dos eventos que podem vir a acordar um dos demais membros do conjunto. Dessa forma, todos os processos do conjunto vão ficar eternamente bloqueados.

Exemplo:

P0	P1
`DOWN(S)`	`DOWN(Q)`
`DOWN(Q)`	`DOWN(S)`
…	…
`UP(S)`	`UP(Q)`
`UP(Q)`	`UP(S)`

Problemas com semáforos: Imagina que P0 executa `DOWN` (S) e depois P1 executa `DOWN` (Q). Quando P0 for executar `DOWN` (Q) ele deve esperar até que P1 execute UP(Q). Da mesma forma, quando P1 executar `DOWN` (S) ele ficará bloqueado até que P0 execute UP(S). Como estas operações não serão executadas, P0 e P1 estão em estado de *deadlock*.

Exercicios

1. The following pseudocode protects a critical section if the semaphore variable S is initially 1.

down(&S); /* entry section or gatekeeper */
<**critical section**>
up(&S); /* exit section */
<**remainder section**>

What happens if S is initially 0 in the previous example? What happens if S is initially 8? Under what circumstances might initialization to 8 prove useful?

2. What happens in the following pseudocode if the semaphores S and Q are both initially 1? What about other possible initializations?

Process 1 executes:	**Process 2 executes**:
for(; ;){	for(; ;) {
down(&S);	down(&Q);
a;	b;
up(&Q);	up(&S);
}	}

3. Implemente o problema do produtor consumidor em Java e faça uso do **Synchronized** na solução.

4. Alphonse and Gaston are friends, and great believers in courtesy. A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow. Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time. This example application, Deadlock, models this possibility:

```
public class Deadlock {
    static class Friend {
        private final String name;
        public Friend(String name) {
            this.name = name;
        }
        public String getName() {
            return this.name;
        }
```

```
        public synchronized void bow(Friend bower) {
            System.out.format("%s: %s has bowed to me!%n",
                    this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s has bowed back to me!%n",
                    this.name, bower.getName());
        }
    }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

Atividade: Implemente este programa (Java) e descubra porque e como é possível a ocorrência de deadlock.

Referências Bibliográficas

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

[2] TANENBAUM, A. S. *Sistemas Operacionais Modernos*. 2ª Edição, Prentice-Hall, 2003.

[3] E.W. DIJKSTRA. "Go To Statement Considered Harmful, Communications of the ACM". Vol. 11, 1968.