Computação Concorrente (MAB-117)
Padrões de uso de semáforos em problemas de concorrência

Prof. Silvana Rossetto

[1]Departamento de Ciência da Computação (DCC)
Instituto de Matemática (IM)
Universidade Federal do Rio de Janeiro (UFRJ)
Abril de 2012

1. Padrões de uso de semáforos

Existem alguns padrões de uso de semáforos para resolver problemas de sincronização. Desenvolver a habilidade de reconhecer e aplicar esses padrões é o primeiro passo para compreender e escrever programas concorrentes usando semáforos [1].

1.1. Exclusão mútua em seções críticas

Como já vimos, um dos padrões de uso de semáforos é para resolver o problema da seção crítica. O semáforo `s` é inicializado com valor 1. A entrada na seção crítica é implementada executando a operação `P(&s)` e a saída da seção crítica é implementada executando a operação `V(&s)`.

1.2. Alocação de recursos

Outro padrão de uso de semáforos é para coordenar a alocação de um recurso, ou de várias réplicas de um mesmo recurso, para threads concorrentes. Considere o cenário de duas ou mais threads concorrendo pelo uso de um recurso (com ou sem réplicas). Quando o recurso (ou nenhuma de suas réplicas) está disponível, a thread deve esperar até que o recurso (ou uma de suas réplicas) seja liberado por outra thread. Um **semáforo contador** pode ser usado para resolver esse problema, como ilustrado abaixo:

```
sem_t s;
sem_init(&s, 0, M) //M igual a número de réplicas do recurso
T1:                    T2:                    T3:
P(&s);                 P(&s);                 P(&s);
//seção crítica        //seção crítica        //seção crítica
V(&s);                 V(&s);                 V(&s);
```

1.3. Alocação de recursos com variável contadora

Outra alternativa para o problema de alocação de recursos faz uso de uma variável contadora `count` para manter a informação sobre o número de recursos (ou réplicas do recurso) disponíveis. Quando o valor de `count` é maior que zero, o recurso está disponível, e quando é igual ou menor que zero, o recurso não está disponível. O valor de `count` é decrementado quando um recurso é alocado para uma thread e incrementado quando o recurso é devolvido.

Outra variável compartilhada (`waiting`) é usada para manter a informação do número de threads aguardando pelo recurso. Quando uma thread deseja usar o recurso ela checa o valor de `count`. Se $count \leq 0$, a thread incrementa o valor de `waiting` e então bloqueia em um semáforo binário. Quando uma thread libera o recurso, ela checa o

valor de `waiting`. Se $waiting > 0$, uma thread bloqueada é sinalizada, senão a variável `count` é incrementada. O código abaixo ilustra esse padrão de uso de semáforos:

```
int count = N,  waiting = 0;
sem_t em, recDisp;

sem_init(&em, 0, 1);
sem_init(&recDisp, 0, 0);

//código executado por cada thread
1: P(&em);
2: if(count > 0) {
3:   count--;
4:   V(&em);
5: } else {
6:   waiting++;
7:   V(&em);
8:   P(&recDisp);
9: }
10: //seção crítica (usa o recurso)
11: P(&em);
12: if(waiting > 0) {
13:   waiting--;
14:   V(&recDisp);
15: } else count++;
16: V(&em);
```

As operações sobre o semáforo binário `em` garante a exclusão mútua no acesso às variáveis `count` e `waiting`. O semáforo `recDisp` é usado para manter uma fila das threads bloqueadas esperando pelo recurso.

Nesse exemplo, podemos ver dois subpadrões importantes no uso de semáforos: **Entra-e-Testa** e **Sai-antes-Bloquear**. Normalmente, uma thread entra na seção crítica (SC) para testar a condição que envolve variáveis compartilhadas (ex., linhas 1-4 do código mostrado acima). Mas antes de se bloquear em uma operação `P()` a espera de um recurso, a thread deve sair da SC para evitar a ocorrência de deadlock (se a thread se bloquear dentro da SC ela pode impedir que outra thread execute a SC e, consequentemente, dê condição para a própria thread ser desbloqueada).

Outro subpadrão de uso de semáforos mostrado no exemplo de código acima é o de **filas condicionais**. Um semáforo pode ser usado para manter uma fila de threads que esperam por uma determinada condição. No exemplo acima, o semáforo `recDisp` é usado para manter uma fila de threads que estão esperando por um recurso (linhas 1-16). Ele é inicializado com o valor 0. Todas as chamadas à operação `recDisp.V()` são executadas quando há ao menos uma thread esperando pelo recurso, e todas as chamadas à operação `recDisp.P()` irão bloquear a thread corrente.

1.4. Barreiras

Vários problemas computacionais são resolvidos usando algoritmos iterativos que sucessivamente computam aproximações melhores para uma resposta procurada. O algoritmo termina quando a resposta é encontrada ou, no caso de vários exemplos de algoritmos numéricos, quando a resposta final convergiu. Normalmente esses algoritmos manipulam

um vetor de valores e cada iteração executa a mesma computação sobre todos os elementos do vetor. Então é possível usar várias threads para computar partes disjuntas da solução de forma concorrente/paralela [2].

Um requisito básico para a maioria dos algoritmos paralelos iterativos é que cada iteração depende dos resultados da iteração anterior. Para garantir que as threads trabalhem sempre em fase (na mesma iteração) é necessário usar um tipo de sincronização chamada **sincronização por barreira**: um ponto de parada é implementado no final de cada iteração e apenas depois de todas as threads passarem por esse ponto é que a computação pode passar para a fase seguinte. Em outras palavras, uma *barreira* é um tipo de sincronização coletiva que suspende a execução das threads de um aplicação em um dado ponto do código e somente permite que as threads prossigam quando todas elas tiverem chegado aquele ponto.

Uma maneira simples de implementar uma barreira é usar um contador que é inicializado com o número total de threads envolvidas. Cada thread decrementa o contador após alcançar a barreira e então se bloqueia esperando o contador chegar a zero. Quando o contador chega a zero todas as threads são desbloqueadas. O código abaixo mostra uma implementação de sincronização por barreira usando semáforos:

```
int cont = NTHREADS;
sem_t em;
sem_t continua[NTHREADS];

sem_init(&em, 0, 1);
for(t=0; t<NTHREADS; t++) {
   sem_init(&continua[t], 0, 0);
}

//código executado por cada thread
for (i=0; i<NITER; i++) {
    //executa a computacao de cada iteracao...

    //sincroniza com as demais threads
    P(&em); //entrada na SC
    cont--;
    if (cont == 0) {
       cont=NTHREADS;
       for (t=0; t<NTHREADS; t++) {
          V(&continua[t]);
       }
    }
    V(&em); //saida da SC
    P(&continua[tid]);
 }
```

A variável compartilhada `cont` é usada para contabilizar o número de threads que já chegaram no ponto de encontro da barreira. O semáforo `em` é usado para controlar o acesso à variável `cont` e o vetor se semáforos `continua` é usado para bloquear as threads na barreira. O semáforo `em` (usado para exclusão mútua) é inicializado com 1. Os semáforos de bloqueio das threads são inicializados com valor 0. As threads executam um número fixo de iterações. A cada iteração, todas as thread decrementam a variável `cont`, para registrar sua chegada à barreira, e se bloqueiam no semáforo `continua[tid]`.

A última thread a chegar na barreira é responsável por liberar todas as threads para a próxima iteração.

2. O problema do produtor/consumidor

O **problema do produtor/consumidor** é um exemplo clássico usado para mostrar o uso de semáforos para implementar as duas formas de sincronização: por exclusão mútua e por condição. O problema consiste em uma estrutura de dados compartilhada (ex., *buffer*, arquivo, tabela de um banco de dados, etc.) que armazena itens gerados por **threads produtoras** e consumidos (retirados da estrutura de armazenamento) por **threads consumidoras**. A sincronização por exclusão mútua é necessária para garantir acesso exclusivo à estrutura de dados usada, enquanto a sincronização por condição é necessária para garantir que não ocorra nem sobreposição de itens (produtor tentando inserir novo item quando a estrutura de dados já está cheia), nem remoção de itens inválidos (consumidor tentando retirar um item quando a estrutura de dados está vazia).

O pseudocódigo abaixo mostra como semáforos podem ser usados para resolver os problemas de sincronização do problema produtor/consumidor.

```
program Prod-Cons {
   const int tam_buffer = N;
   T_item buffer[N];
   semaphore EM=1, CHEIO=0, VAZIO=N;

   void Produtor() {
      while(true) {
         produz_item();  //fora da secao critica
         P(VAZIO);
         P(EM);
         insere_item(buffer);
         V(EM);
         V(CHEIO);
      }
   }

   void Consumidor() {
         P(CHEIO);
         P(EM);
         retira_item(buffer);
         V(EM);
         V(VAZIO);
         consome_item(); //fora da secao critica
   }

   void main() {
     //inicia threads produtoras e consumidoras
   }
}
```

3. O problema dos leitores e escritores

O problema dos *leitores e escritores* é outro exemplo de problema clássico de concorrência. Uma área de dados (ex., arquivo, bloco da memória, tabela de uma banco de dados) é compartilhada entre diferentes threads. As **threads leitoras** apenas lêem o conteúdo

da área de dados, enquanto as **threads escritoras** apenas escrevem dados nessa área.As seguintes condições são definidas para o problema dos leitores/escritores:

1. Os leitores podem ler simultaneamente uma região de dados compartilhada.
2. Apenas um escritor pode escrever a cada instante em uma região de dados compartilhada.
3. Se um escritor está escrevendo, nenhum leitor pode ler a mesma região de dados compartilhada.

As interações leitores/escritores ocorrem frequentemente em sistemas reais [3]. Por exemplo, em um sistema de reservas de passagens aéreas, um grande número de usuários pode inspecionar concorrentemente os assentos disponíveis, mas um usuário que está reservando um assento deve ter acesso exclusivo à base de dados.

O problema dos leitores/escritores têm diferentes variações, dependendo da prioridade dada às operações de leitura e escrita. Quando a leitura tem maior prioridade, as operações de escrita são retardadas enquanto houver leitores querendo ler a região de dados compartilhada. Quando a escrita tem maior prioridade, as operações de leitura são retardadas caso exista algum escritor querendo escrever na região de dados compartilhada. Uma solução para o problema, com prioridade para leitura, é mostrada abaixo [3]:

```
int contLeit = 0;
sem_t em, escrita;

sem_init(&em, 0, 1);
sem_init(&escrita, 0, 1);

void leitor() {
   while(1) {
      P(&em);
      contLeit++;
      if(contLeit == 1) P(&escrita);
      V(&em);
      // seção crítica: leituras...
      P(&em);
      contLeit--;
      if(contLeit == 0) V(&escrita);
      V(&em);
   }
}

void escritor() {
   while(1) {
      P(&escrita);
      //seção crítica: escritas...
      V(&escrita);
   }
}
```

O semáforo `escrita` controla o acesso à seção crítica do código para operações de escrita. O semáforo `em` proteje o acesso à variável compartilhada `contLeit`, a qual contabiliza o número de leitores ativos na seção crítica de operações de leitura. Um escritor aloca o semáforo `escrita` cada vez que entra na sua seção crítica e desaloca esse semáforo quando sai da seção crítica. Isso garante que sempre haverá no máximo

um escritor alterando a região compartilhada a cada instante. Por outro lado, apenas o primeiro leitor a entrar na seção crítica aloca o semáforo `escrita`, e apenas o último leitor a deixar a seção crítica desaloca esse semáforo. Isso significa que assim que um leitor aloca o semáforo `escrita`, um número ilimitado de leitores pode executar a seção crítica de leitura sem bloqueio. Essa solução pode resultar em *starvation* das threads escritoras.

Referências

[1] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading*. Wiley, 2006.

[2] G. Andrews. *Concurrent Programming — Principles and Practice*. Addison-Wesley, 1991.

[3] R. E. Bryant and D. R. O'Hallaron. *Computer Systems - A Programmer's Perspective*. Prentice-Hall, 2 edition, 2010.