O Problema dos Leitores/Escritores Usando Programação Concorrente com ADA95

Alexandre Keller Albalustro
Marcos Banheti Rabello Vallim

Abstract

This paper presents a solution to the readers/writers problem using concurrent programming with ADA. The problem is an abstraction of access to databases, very important in computer systems, where a database may need to be protected by some solution that ensures consistency. To do this, we have shown some solutions using rendezvous in Ada and how well these solutions work.

Resumo

Este artigo apresenta algumas soluções para o problema dos leitores/escritores usando programação concorrente com ADA95. O problema é uma abstração do acesso à base de dados, muito importante em sistemas de computadores, onde uma base de dados necessita ser protegida por alguma solução que assegure sua consistência. Para fazer isto, são mostradas algumas soluções usando o mecanismo de rendezvous em ADA95.

PALAVRAS-CHAVES:

leitores/escritores, exclusão mútua, rendezvous, ADA95, programação concorrente.

1. INTRODUÇÃO

O problema dos leitores/escritores é um caso clássico de sincronização, onde tarefas - leitores e escritores - competem pelo acesso a uma base de dados compartilhada. [BEN90]

Neste problema, diversos leitores podem concorrentemente acessar a base de dados, entretanto, a atualização desta deve ser feita sob acesso exclusivo. A exclusão mútua para os escritores é necessária para assegurar a consistência da base de dados, de maneira que somente um escritor possa acessar a base de dados por vez.

O problema pode ser resolvido considerando-se os seguintes casos:

a) Solução priorizando acesso aos leitores;
b) Solução priorizando acesso aos escritores;
c) Solução com igualdade de prioridade aos leitores e escritores.

2. LINGUAGEM DE PROGRAMAÇÃO CONCORRENTE ADA95

A linguagem de programação Ada95 foi escolhida devido à sua flexibilidade e capacidade para modelagem de concorrência, mesmo quando

Alexandre Keller Albalustro é Engenheiro do Departamento de Automação & Sistemas, Universidade Federal de Santa Catarina. E-mail: alex@lcmi.ufsc.br

Marcos Banheti Rabello Vallim é professor no Centro Federal de Educação Tecnológica do Paraná, Unidade de Cornélio Procópio. E-mail: mvallim@lcmi.ufsc.br

comparada a outras linguagens tais como *Java, SR* ou *Oberon*. Ada foi desenvolvida a pedido do Departamento de Defesa dos Estados Unidos (DoD) como a linguagem-padrão para desenvolvimento de sistemas de missão crítica, inclusive sistemas militares.

Ao longo da década de 80, importantes sistemas comerciais foram fortemente encorajados para serem implementados em Ada.

Embora persistam certos mitos com relação à linguagem Ada, que é considerada uma linguagem "militar", a verdade é que Ada é amplamente utilizada fora desse círculo.

As aplicações são as mais diversas possíveis, tais como sistemas *PABX*, projeto de circuitos integrados, dispositivos médicos, controle de fábricas, *GPS*, sistemas embarcados de aviões [AONIX97].

A versão utilizada neste trabalho é a Ada95, que é uma extensão orientada a objetos da linguagem Ada tradicional, conhecida também como Ada83.

2.1 Comunicação e Sincronização em Ada

O mecanismo de sincronização em Ada é chamado ***rendezvous*** [Ben90]. Tal mecanismo permite o encontro de duas tarefas, de maneira que elas possam comunicarem-se e sincronizarem-se.

Ada implementa um modelo de sincronização assimétrico, em que a tarefa chamadora (*caller*) deve conhecer a entrada pertencente à tarefa servidora (*server*) e também a sua identidade. A tarefa servidora não necessita conhecer a identidade da tarefa chamadora.

Uma tarefa é uma linha de execução independente (*thread*) de um programa Ada, e é executada concorrentemente com outras tarefas existentes, e que podem somente serem sincronizadas por um mecanismo de ***rendezvous***.

3. SOLUÇÃO DO PROBLEMA DOS LEITORES/ESCRITORES

3.1 Solução priorizando os Leitores

Uma tarefa pode ser usada para arbitrar o acesso à base de dados (tarefa de controle de acesso), e outras chamadas leitores e escritores (tarefas chamadoras) irão competir pelo acesso à base de dados. Salienta-se aqui a importância da tarefa de controle de acesso, a qual possui como missão mediar o acesso das tarefas chamadoras.

```
Select
when nw = 0  =>
accept readers_in do
                nr := nr  + 1;
end readers_in;
or when (nr = 0 and nw = 0)  =>
accept writers_in do
                nw := nw  + 1;
end writers_in;
accept readers_out do
                nr := nr  – 1;
end readers_out;
accept writers _out do
                nw := nw – 1;
end writers _out;
end Select;
```

Fig. 1 – Solução priorizando os leitores

Os leitores não competem uns com os outros e um escritor exclui todas as outras tarefas. Para fornecer uma solução que priorize os leitores, devem ser adicionados comandos de guarda adequados nas entradas do controle de acesso para expressar esta preferência. Para fazer isto, podem ser utilizadas duas variáveis para contabilizar o número de tarefas chamadoras acessando a base de dados.

Definindo **(nr)** como número de leitores e **(nw)** como número de escritores, então pode-se construir os comandos de guarda que assegurem a preferência aos leitores.

Como pode ser visto na figura 1, a entrada para os escritores é mais proibitiva do que a entrada para os leitores. Um escritor necessita esperar até que nenhum leitor ou escritor esteja acessando a base de dados. Entretanto, um leitor somente precisa esperar para que nenhum escritor esteja acessando a base de dados.

Embora a solução mostrada na figura 1 pareça estar totalmente correta, ela, entretanto, não fornece aos leitores uma preferência tão forte quanto desejada. Caso esta solução não esteja adequada à especificação desejada, deve-se utilizar outra variável inteira para expressar a quantidade de leitores que desejam acessar a base de dados.

No entanto, um escritor somente obterá acesso se nenhum leitor estiver esperando na fila e, claro, se não houver nenhum leitor ou escritor acessando a base de dados.

Definindo **(rr)** como número de leitores desejando acessar a base de dados, a nova solução baseada neste algoritmo revisado é mostrada na figura 2. Desta maneira, foi obtida uma solução mais correta que a anterior.

```
Select
when nw = 0  =>
accept readers_in do
        nr := nr  + 1;
        rr := rr  – 1;
end readers_in;
or when (nr = 0 and nw = 0 and rr = 0)  =>
accept writers_in do
        nw := nw  + 1;
end writers_in;
accept readers_out do
        nr := nr  – 1;
end readers_out;
accept writers_out do
        nw := nw – 1;
end writers_out;
accept readers_waiting do
        rr := rr  - 1;
end readers_waiting;
end Select;
```

Fig. 2 – Nova solução priorizando os leitores.

3.2 Solução Priorizando Escritores

Para fornecer uma solução priorizando os escritores, é necessário modificar a solução revisada dos leitores, de forma que o contador **(nr)** não seja incrementado se um escritor estiver acessando a base de dados ou se existirem outros esperando para acessar. Para fazer isto, deve ser modificado o comando de guarda para a entrada dos leitores.

Definindo **(ww)** como o número de escritores desejando acessar a base de dados, o novo comando de guarda para a entrada dos leitores é mais restritivo do que aquele da solução anterior.

O contador **(ww)** se faz necessário porque ele indica o número de escritores que estão esperando, de tal forma que se **(ww)** for maior que zero, nenhum leitor poderá acessar a base de dados. Desta maneira, enquanto houver escritores ativos, eles terão total preferência sobre os leitores. Esta solução é mostrada na figura 3.

```
Select
when (nw = 0 and ww = 0)  =>
accept readers_in do
                nr := nr + 1;
end readers_in;
or when  (nr = 0 and nw = 0)   =>
accept writers_in do
                nw := nw + 1;
                ww := ww – 1;
end writers_in;
accept readers_out do
                nr := nr  – 1;
end readers_out;
accept writers_out do
                nw := nw + 1;
end writers_out;
accept writers_waiting do
                rr := rr  + 1;
end writers_waiting;
end Select;
```

Fig. 3 – Solução priorizando os escritores

3.3 Prioridade Igual para Leitores e Escritores

Se um leitor estiver acessando a base de dados e nenhum escritor chegar, todos os outros leitores poderão também acessar a base de dados. Mas, assim que um escritor desejar escrever na base de dados, nenhum outro leitor poderá acessá-la.

```
Select
when ((nw = 0 and ww = 0) or
(nw = 0 and ww > 0 and fflag = Wflag)) =>
accept readers_in do
        nr := nr  + 1;
        rr := rr  – 1;
        if (rr  =  0) then
            fflag := Rflag;
        end if;
end readers_in;
or when ( (nw = 0 and nr = 0 and rr = 0) or
(nw = 0 and nr = 0 and rr > 0 and fflag = Rflag)) =>
accept writers_in do
        nw := nw  + 1;
        ww := ww – 1;
        fflag := Wflag;
end writers_in;
accept readers_out do
        nr := nr  – 1;
end readers_out;
accept writers_out do
        nw := nw – 1;
end writers_out;
accept readers_waiting do
        rr := rr  + 1;
end readers_waiting;
accept writers_waiting do
        ww := ww  + 1;
end writers_waiting;
end Select;
```

Fig. 4 – Solução que fornece a mesma prioridade aos leitores e escritores

Os novos leitores terão que esperar até que o escritor finalize sua operação na base de dados. Certamente, neste caso, os comandos de guarda para o controle de acesso devem ser mais justos do que aqueles das soluções anteriores. Por outro lado, a especificação de justiça torna os comandos de guarda um pouco mais complexos, mas garantem que escritores e leitores tenham a mesma prioridade para acessar a base de dados. A figura 4 mostra um resumo da idéia mencionada acima.

Foi usado um sinalizador - uma variável inteira - para expressar a última tarefa que deixou a base de dados. A justiça entre leitores e escritores é portanto fortemente garantida. Porém, se um escritor acabou de acessar a base de dados, o valor do sinalizador será **wflag** (uma constante inteira). Desta maneira, nenhum outro escritor

poderá acessar a base de dados, a menos que não existam leitores esperando na fila.

O teste *if* para a entrada de leitores é necessário porque ele bloqueará qualquer outro caso em que um escritor chegar, mesmo que existam leitores acessando a base de dados.

4. IMPLEMENTAÇÃO DO CÓDIGO-FONTE

Há duas unidades nesse trabalho, **main.ada** e **rw.ada** (apêndice A). A primeira unidade contém as tarefas leitores e escritores, as quais são dinamicamente criadas e alocadas dentro de um *array*. Estas tarefas devem passar através de uma tarefa de controle de acesso - contida em **rw.ada** - para acessar a base de dados.

O código para cada leitor ou escritor é muito simples. Assim que estas são criadas, elas são atrasadas por um tempo aleatório – um valor entre 3 e 6 segundos, retornando a partir da função *RandonTime* – antes que as tarefas chamem qualquer entrada do controle de acesso.

A solução para cada controle de acesso (seções 3.1, 3.2, 3.3) foi codificada em *packages* Ada, como pode ser visto na segunda unidade, **rw.ada**.

Dentro de cada *package*, há um algoritmo, conforme foi mostrado na terceira seção deste artigo, implementado para resolver uma desejada especificação.

Basicamente o controle de acesso é um laço infinito que contém uma declaração *select* para aceitar chamadas para uma de suas diversas entradas.

Devido à alternativa *terminate* na declaração de seleção, o controle de acesso pode terminar se não existir nenhuma chamada para suas entradas. O outro *package* de **rw.ada**, chamado *out_user*, simplesmente providencia uma interface de comunicação entre o usuário e o programa.

5. CONCLUSÃO

Este artigo fornece algumas possíveis soluções para o problema dos leitores/escritores usando ***rendezvous*** em Ada. A utilização deste mecanismo, conforme mostrado, simplifica a implementação de soluções deste problema.

Outros mecanismos de sincronização como monitores ou semáforos, poderiam ter sido usados também. Entretanto, eles teriam que ser implementados através de ***rendezvous***, o que introduziria complicações adicionais ao código, sem nenhum ganho de desempenho ao programa. Assim sendo, é altamente recomendável o uso do mecanismo de ***rendezvous*** puro, como utilizado com sucesso na implementação apresentada.

Os resultados das simulações mostram que os algoritmos desenvolvidos para cada caso considerado correspondem às especificações desejadas, estando amplamente validados.

6. REFERÊNCIAS

[Ben90] M. Bem-Ari. Principles of Concurrent and Distribuited Programming. Prentice Hall International - Series in Computer Science, Hemel Hempstead, 1990.

[Gra91] G.R. Andrews. Concurrent Programming - Principles and Practice. Benjamin/Cummings Publishing Company, Menlo Park, CA, 1991.

[AONIX97] Is Ada Good for my Bussiness? AONIX White-Paper, December, 1997.

[AONIX] Press Release, November, 1997, disponível no AONIX web site, localizado em http://www.aonix.com

RECONHECIMENTOS

Este trabalho foi realizado para a disciplina de Engenharia de Sistemas de Tempo-Real II, do Curso de Pós-Graduação em Engenharia Elétrica. O compilador usado neste trabalho foi ObjectAda Special Edition 7.1 for Windows da AONIX Enterprise (San Diego, CA- USA).

Apêndice A: rw.ada

```
— Package for readers' preference solution —

with out_user; use out_user;
package RPS_readers_writers is
  task type RPS_control_access is
    entry readers_in;
    entry readers_out;
    entry readers_waiting;
    entry writer_in;
    entry writer_out;
  end RPS_control_access;
end RPS_readers_writers;

package body RPS_readers_writers is
  task body RPS_control_access is
    nr, nw, rr : Integer:=0;
    begin
      loop
      select
        when (nw = 0) =>
          accept readers_in do
            nr:=nr + 1;
            rr:=rr – 1;
            write_msg_count("> Reader accessing the database:",nr);
          end readers_in;
      or when ((nw = 0) and (nr = 0) and (rr = 0)) =>
          accept writer_in do
            nw:=nw+1;
            write_msg("> Writer accessing the database");
          end writer_in;
      or accept readers_out do
            nr:=nr – 1;
            write_msg("> Reader out of the database");
          end readers_out;
      or accept writer_out do
            nw:=nw - 1;
            write_msg("> Writer out of the database");
          end writer_out;
    or accept readers_waiting do
            rr:=rr + 1;
          end readers_waiting;
    or terminate;
    end select;
    end loop;
  end RPS_control_access;
end RPS_readers_writers;

— Package for writers' preference solution —

with out_user; use out_user;
package WPS_readers_writers is
  task type WPS_control_access is
    entry readers_in;
    entry readers_out;
    entry writer_in;
    entry writer_out;
    entry writers_waiting;
  end WPS_control_access;
end WPS_readers_writers;

package body WPS_readers_writers is
  task body WPS_control_access is
  nr, nw, ww : Integer:=0;
  begin
   loop
   select
      when ((nw = 0) and (ww = 0)) =>
        accept readers_in do
          nr:=nr+1;
          write_msg_count("> Reader accessing the database:",nr);
        end readers_in;
     or when ((nw = 0) and (nr = 0)) =>
      accept writer_in do
        nw:=nw+1;
        ww:=ww-1;
        write_msg("> Writer accessing the database");
       end writer_in;
     or accept readers_out do
          nr:=nr-1;
          write_msg("> Reader out of the database");
        end readers_out;
  or accept writer_out do
          nw:=nw-1;
          write_msg("> Writer out of the database");
        end writer_out;
     or accept writers_waiting do
          ww:=ww+1;
          write_msg_count("> Writer wishing to access the database:",ww);
        end writers_waiting;
     or terminate;
     end select;
     end loop;
  end WPS_control_access;
end WPS_readers_writers;

— Package for the same priority for the readers and the writers —

with out_user; use out_user;
package SPRW_readers_writers is
  task type SPRW_control_access is
    entry readers_in;
    entry readers_out;
    entry writer_in;
    entry writer_out;
    entry readers_waiting;
    entry writer_waiting;
  end SPRW_control_access;
end SPRW_readers_writers;

package body SPRW_readers_writers is
  task body SPRW_control_access is
  nr, nw, rr, ww : Integer := 0;
  Rflag : Constant Integer := 1;
  Wflag : Constant Integer := 2;
  fflag : Integer := Rflag;
  begin
    loop
    select
      when ((nw = 0 and ww = 0) or (nw = 0 and ww > 0 and fflag =
Wflag)) =>
        accept readers_in do
          nr:=nr+1;
          rr:=rr-1;
          if rr=0 then
            fflag := Rflag;
            end if;
            write_msg_count("> Reader accessing the database:",nr);
        end readers_in;
        or when ((nw = 0 and nr = 0 and rr = 0) or
              (nw = 0 and nr = 0 and rr > 0 and fflag = Rflag)) =>
        accept writer_in do
          nw:=nw+1;
          fflag := Wflag;
          ww:=ww-1;
          write_msg("> Writer accessing the database");
        end writer_in;
     or accept readers_out do
          nr:=nr-1;
          write_msg("> Reader out of the database");
      end readers_out;
      or accept writer_out do
          nw:=nw-1;
          write_msg("> Writer out of the database");
        end writer_out;
    or accept readers_waiting do
          rr:=rr+1;
          write_msg_count("> Reader wishing to access the database",rr);
        end readers_waiting;
     or accept writer_waiting do
          ww:=ww+1;
          write_msg_count("> Writer wishing to access the database",ww);
        end writer_waiting;
    or terminate;
    end select;
    end loop;
  end SPRW_control_access;
end SPRW_readers_writers;
```