Programação Concorrente

Locks

Prof. Eduardo Alchieri

Locks

- **Variáveis do tipo trava (lock)**
- **Lock:** É um mecanismo de sincronização de processos/threads, em que processos/threads devem ser programados de modo que seus efeitos sobre os dados compartilhados sejam **equivalentes serialmente**.
- Se for sabido que cada uma de várias threads tem o mesmo efeito correto quando executada sozinha, então podemos inferir que, se essas threads forem executadas uma por vez, em alguma ordem, o efeito combinado também será correto.
- Uma intercalação das operações das threads em que o efeito combinado é igual ao que seria se as threads tivessem sido executadas uma por vez, em alguma ordem, é uma intercalação equivalente serialmente.

Locks

- **Variáveis do tipo trava (lock)**
  - Quando dizemos que duas threads distintas tem o mesmo efeito, queremos dizer que as operações de leitura sobre variáveis (exemplo, saldo de contas bancárias) retornam os mesmos valores e que essas variáveis tem os mesmos valores finais.
  - O uso de equivalência serial como critério para execução concorrente correta evita a ocorrência de atualizações perdidas ou recuperações inconsistentes.
  - Um exemplo simples de mecanismos para disposição em série é o caso de locks (travas) exclusivos. Nesse esquema, um processo tenta impedir o acesso (travar) a qualquer variável compartilhada que esteja utilizando
  - Se outro processo solicitar acesso a uma variável que está bloqueada (travado), o processo deverá esperar até que a variável seja destravada.

Locks

- **Variáveis do tipo trava (lock)**
  - Cria-se uma variável única compartilhada (variável de travamento), cujo valor pode assumir 0 ou 1
  - O valor em 0 significa que não há nenhum processo executando a sua região crítica, e o valor em 1 significa que algum processo está executando sua região crítica
  - Problema: dois processos (A e B) tentam entrar em suas regiões críticas simultaneamente, então A, por exemplo, pega o valor da VT (variável de travamento) em 0, porém A pode não conseguir atualizar o valor de VT antes que B o acesse, o que vai gerar condição de corrida

Locks

- **Primeira tentativa para implementar um lock**

Variável global: *em_uso*: boolean initial false;

A variável global indica se alguma região crítica está em uso ou não.

Código de um processo:

```
. . .
loop
    exit when ¬em_uso
endloop;
em_uso:=true;
REGIÃO CRÍTICA;
em_uso:= false;
. . .
```

A solução não é correta, pois os processos podem chegar à conclusão "simultaneamente" que a entrada está livre (*em_uso*=false). Isto é, os dois processos podem ler (testar) o valor de *em_uso* antes que essa variável seja feita igual a *true* por um deles.

Locks

- **Chaveamento Obrigatório**
  - Mais uma forma implementada por software, que utiliza a variável inteira *turn*
    - A variável inteira *turn* estabelece de quem é a vez de entrar na região crítica

```
while (TRUE) {
        while (turn != 0) /*espera*/;
        regiao critica ( );
        turn = 1;
        regiao não-critica ( );
} (a) Processo 0
```

```
while (TRUE) {
        while (turn != 1)  /*espera */;
        regiao critica ( );
        turn = 0;
        regiao não-critica ( );
} (b) Processo 1
```

Locks

- **Segunda tentativa para implementar um lock**

Variável global: *vez*: integer initial 1;

Esta variável indica de quem é a vez, na hora de entrar na região crítica.

Código de um processo:

```
. . .
loop
    exit when vez=eu
endloop;
REGIÃO CRÍTICA;
vez:= outro;
. . .
```

Este algoritmo garante exclusão mútua, mas obriga a alternância na execução

Locks

- **Chaveamento Obrigatório**
  - Não é uma boa ideia quando um dos processos é muito mais lento que o outro
    - Esta solução requer a entrada estritamente alternada de dois processos em suas regiões críticas

Locks

- **Terceira tentativa para implementar um lock**

Variável global: *quer*: array[2] of boolean initial false;

Se *quer*[i] é *true*, isto indica que o processo Pi (i∈{1,2}) quer entrar na sua região crítica. Observe que o valor inicial especificado para um *array* (vetor ou matriz) se aplica a todos os elementos desse *array*.

Código de um processo:

```
. . .
loop
    exit when ¬quer[outro]
endloop;
quer[eu]:=true;
REGIÃO CRÍTICA;
quer[eu]:=false;
. . .
```

A solução não assegura exclusão mútua. Repete-se aqui o mesmo problema da tentativa 1, pois cada processo pode chegar à conclusão que o outro não quer entrar e assim entrarem

Locks

- **Quarta tentativa para implementar um lock**

Variável global: *quer*: array[2] of boolean initial false;

É o mesmo algoritmo anterior, porém marcando a intenção de entrar, antes de testar a intenção do outro processo.

Código de um processo:

```
. . .
quer[eu]:=true;
loop
    exit when ¬quer[outro]
endloop;
REGIÃO CRÍTICA;
quer[eu]:=false;
. . .
```

Com este algoritmo a exclusão mútua é garantida, mas, infelizmente, os processos podem entrar em um loop eterno. Isto porque ambos os processos podem marcar "simultaneamente" a intenção de entrar (antes que um deles consiga testar, dentro do loop, se o outro quer entrar ou não). Nesse caso, depois de entrarem no loop, os processos não vão sair mais de lá.

Locks

- **Quinta tentativa para implementar um lock**

Tentativa 5

Variável global: *quer*: array[2] of boolean initial false;

É semelhante ao algoritmo anterior, porém o processo dá a vez para o outro no caso do outro querer entrar.

Código de um processo:

```
            . . .
início:     quer[eu]:=true;
            if quer[outro] then
                { quer[eu]:=false;
                  goto inicio
                };
            REGIÃO CRÍTICA;
            quer[eu]:=false;
            . . .
```

A exclusão mútua é garantida, mas os processos podem ficar dando a vez indefinidamente. Embora difícil de ocorrer na prática, não deixa de ser teóricamente possível.

Locks

- **Solução de Dekker**

Variáveis globais: *quer*: array[2] of boolean initial false;
vez: integer initial 1;

Esta solução foi proposta pelo matemático holandês T. Dekker e discutida por Dijkstra [DIJ 65a]. Trata-se da primeira solução completa para o problema. Conforme já referido, é similar ao algoritmo anterior. A diferença está no uso da variável *vez*, para realizar o desempate (*tie-break*). No caso em que os dois processos entram no bloco entre chaves, só um deles (decidido pelo valor da variável *vez*) dá a vez para o outro.

Locks

- **Solução de Dekker**

Código de um processo:

```
            . . .
início:   quer[eu]:=true;
denovo: if quer[outro] then
            {  if vez=eu then goto denovo;
               quer[eu]:=false;
               loop
                    exit when vez=eu
               endloop;
               goto início
            };
            REGIÃO CRÍTICA;
            quer[eu]:=false;
            vez:=outro;
            . . .
```

O algoritmo satisfaz todas as exigências para a solução ser considerada correta

Locks

- **Solução de Peterson**

Variáveis globais: *quer*: array[2] of boolean initial false;
vez: integer;

Código de um processo:

```
. . .
quer[eu]:= true;
vez:= outro;
loop
    exit when ¬quer[outro] or vez=eu
endloop;
REGIÃO CRÍTICA;
quer[eu]:= false;
. . .
```

Locks

- **Instrução TSL (test and set lock)**
  - Implementada com auxílio de hardware esta instrução funciona da seguinte maneira:
    - Lê o conteúdo da memória, a palavra *lock*, no registrador RX e, então, armazena um valor diferente de zero no endereço de memória *lock*
      - Estas operações são indivisíveis: a CPU que esta executando impede o acesso ao barramento de memória
    - Quando termina de executar a região crítica, o processo volta a colocar o valor de *lock* em zero

Locks

Variável global: *is_free*: boolean initial true;

Código de um processo:

```
. . .
loop
    exit when TS(is_free)
endloop;
REGIÃO CRÍTICA;
is_free:= true;
. . .
```

Locks

- **Exclusão mútua com n processos**
  - Uma solução para n processos é correta se assegura os seguintes requisitos:
    - Garantia de exclusão mútua
    - Garantia de progresso para os processos
    - Garantia de tempo de espera limitado
  - Algoritmo de Dijkstra
    - Não garante espera limitada
      - Outras soluções posuem esta propriedade

Locks

- Algoritmo de Dijikstra

Variáveis globais: *quer*, *dentro*: array[n] of boolean;
vez: integer;

Os dois vetores são inicializados com *false*. Os elementos *quer*[i] e *dentro*[i] são alterados apenas pelo processo i, $1 \leq i \leq n$. A variável *vez* assume valores entre 1 e n e seu valor inicial é irrelevante. Cada processo utiliza uma variável local k.

Locks

Código do processo i:

```
            . . .
            quer[i]:=true;
inicio:     loop
                dentro[i]:=false;
                if ¬quer[vez] then vez:=i;
                exit when vez=i
            endloop;
            dentro[i]:=true;
            for k:=1 to n st k≠i
                if dentro[k] then goto inicio
            endfor;
            REGIÃO CRÍTICA;
            quer[i]:=false;
            dentro[i]:=false;
            . . .
```

Locks

- Este algoritmo garante exclusão mútua: i só entra na região crítica se após fazer dentro[i]=true encontra os demais processos com “dentros” = false;
- No entanto, espera ilimitada não é garantida, pois um processo pode ficar tentando indefinidamente entrar na região crítica.
  - Outras soluções tentam resolver este problema
    - Algoritmo de Eisenberg e McGuire
    - Algoritmo de Peterson para n processos
    - Etc...

Locks

- Solução vistas até aqui não são implementadas na prática
  - O teste contínuo do valor de uma variável, aguardando que ela assuma determinado valor é denominado de **espera ocupada**

Locks

Exercício 4.1 Um especialista em programação concorrente apresentou a seguinte solução para o problema da exclusão mútua entre dois processos:

Variáveis compartilhadas: *quer* : array[2] of boolean initial false;
vez : integer;initial 1;

Processo 1:

```
. . .
quer[1] := true;
loop
    exit when vez=1;
    if ¬quer[2] then vez:=1
endloop;
REGIÃO CRÍTICA;
quer[1] := false;
. . .
```

Processo 2:

```
. . .
quer[2] := true;
loop
    exit when vez=2;
    if ¬quer[1] then vez:=2
endloop;
REGIÃO CRÍTICA;
quer[2] := false;
. . .
```

Responda (justificando a resposta):

(a) Pode haver bloqueio eterno dos dois processos?

(b) Os dois processos podem estar simultaneamente na região crítica?

Locks

- Exercício: Controle de entrada de um parque:
  - A administração de um grande parque deseja controlar através de um sistema computacional as entradas e saídas deste parque a fim de saber, a qualquer momento, quantas pessoas encontram-se no interior do parque. Sabendo que existem duas catracas, uma para entra e outra para sair, construa um protótipo do software a ser usado neste programa.

Locks

- Exercício: Controle do saldo bancário
  - A administração de um banco deseja controlar o saldo de uma conta bancária. Sabendo que existem duas formas de depósito (caixa eletronico e caixa atendimento) e uma forma de saque (caixa atendimento), construa um protótipo do software a ser usado neste controle.

Locks

- Exercício: Controle de um aeroporto
  - Sabendo-se que um aeroporto possui apenas uma pista para pousos e decolagens, construa um protótipo do software a ser usado por um programa que controla o acesso dos aviões a esta pista.

Locks

- Exercício: Controle de uma ponte
  - Uma cidade X possui uma ponte muito antiga que não suporta muito peso. Após medições, ficou estabelecido que apenas um carro poderá estar sobre a ponte a cada momento. Os administradores de X desejam utilizar um software para fazer este controle. Projete este software.

Locks

- Outros exercícios
  - Problema dos macacos
  - Problema dos leitores e escritores

  - Verifique se a solução para estes problemas pode apresentar starvation!