Programação Concorrente e Paralela

Semáforos

Noemi Rodriguez

2016

Primitivas de Concorrência

- necessidade de suporte da linguagem ou do SO para apoiar sincronização entre threads:
  - exclusão mútua
  - condições

Notação Andrews: atomicidade e *await*

- para definir ações atômicas, Andrews introduz a notação $\langle e \rangle$
- para especificar *sincronização*, Andrews introduz a notação:

  $$\langle await(B)S; \rangle$$

  que significa que S só deve começar a ser executado quando B for verdadeira, e que S será executado atomicamente

Primitivas de Concorrência

- necessidade de suporte da linguagem ou do SO para apoiar sincronização entre threads:
  - exclusão mútua
    - primitiva *lock()*

Exemplo de sincronização: produtor/consumidor

- também chamado de problema do buffer limitado
- diversas variantes
  - buffer limitadíssimo!

Exemplo de sincronização: 1 produtor/ 1 consumidor

```
int buf; bool temitem = false;
process Producer {
  int a;
  while (true) {
    /* "produz" valor em a  (regiao nao critica) */
    < await (!temitem);>
    buf = a;
    < temitem = true; >
  }
}
process Consumer {
  int b;
  while (true) {
    < await (temitem);>
    b = buf;
    < temitem = false;>
    /* "consome" valor em b  (regiao nao critica) */
  }
}
```

n produtores / n consumidores

```
int buf; bool temitem = false;
process Producer {
  int a;
  while (true) {
    /* "produz" valor em a  (regiao nao critica) */
    < await (!temitem)
    buf = a;
    temitem = true;
    >
  }
}
process Consumer {
  int b;
  while (true) {
    < await (temitem);
    b = buf;
    temitem = false;
    >
    /* "consome" valor em b  (regiao nao critica) */
  }
}
```

Como implementar o `await` com EM?

- $< S >$ implementado por

```
CSenter(); /* lock! */
S;
CSexit(); /* unlock! */
```

- e $< await(B)S; >$?
  - podemos testar a condição dentro da região crítica?

```
CSenter();
while (!B) (???)
S;
CSexit();
```

await baseado em exclusão mútua

... se a alteração da condição B depender de outro processo entrar na região crítica, nada feito...

await com protocolos de exclusão mútua

... se a alteração da condição B depender de outro processo entrar na região crítica, nada feito...

- podemos tentar:

```
CSenter();
while (!B) (CSexit; CSenter;)
S;
CSexit();
```

  mas teremos grande disputa pela EM...

- aqui poderíamos aplicar o conceito de *back-off*

Implementação de espera por condições

- complicado no caso geral

Semáforos

- exclusão mútua e sincronização por condição
  - Dijkstra, E. W. 1968. The structure of the "THE"-multiprogramming system. *Commun. ACM* 11, 5 (May. 1968), 341–346.
  - proposto para sincronização entre processos do sistema operacional!

  - material baseado no Cap. 4 de FMPDP (Andrews)

Definição

- sintaxe:
  - declaração e inicialização:

    ```
    sem s;
    sem lock  = 1;
    sem forks[5];
    ```

    cada semáforo é associado a um valor inteiro não negativo
  - manipulação: operações `P` e `V`

    ```
    P(s): < await (s > 0) s = s - 1; >
    V(s): < s = s + 1 >
    ```
  - semáforos *binários* e semáforos *gerais*
  - propriedades relativas a *liveness* dependem de especificação de $< await >$

Exclusão Mútua

```
sem em = 1;
process CS [i = 1 to n] {
    P(em);
    a = a + 1;
    V(em);
}
```

- implementada por um semáforo binário

Cooperação

- Voltando ao problema do buffer limitadíssimo:

```
int buf;
sem empty = 1, full = 0;
process Producer {
  int dados;
  while (true) {
    /* produz dados */
    P(empty);
    buf = dados;
    V(full);
  }
}
process Consumer {
  int dados;
  while (true) {
    P(full);
    dados = buf;
    V(empty);
    /* consome dados */
  }
}
```

exemplo de *split binary semaphore*

Cooperação [cont]

- e se tivermos vários produtores e vários consumidores?

```
int buf;
sem empty = 1, full = 0;
process Producer {
  int data;
  while (true) {
    /* produz dados */
    P(empty);
    buf = data;
    V(full);
  }
}
process Consumer {
  int data;
  while (true) {
    P(full);
    data = buf;
    V(empty);
    /* consome data */
  }
}
```

ainda funciona

Cooperação [cont]

- e se tivermos uma estrutura de dados mais complexa?

```
int buf[SIZE]; int nxtfree = 0; int nxtdata = 0;
process Producer {
  int data;
  while (true) {
    /* produz dados */
    buf[nxtfree] = data;
    nxtfree = (nxtfree+1)%SIZE;
  }
}
process Consumer {
  int data;
  while (true) {
    data = buf[nxtdata];
    nxtdata = (nxtdata+1)%SIZE;
    /* consome dados */
  }
}
```

Buffer Limitado - 1 processo de cada tipo

```
int buf[SIZE]; int nxtfree = 0; int nxtdata = 0;
-- uso de semáforos contadores
sem empty = SIZE; sem full = 0;
process Producer {
  int data;
  while (true) {
    /* produz dados */
    P(empty);
    buf[nxtfree] = data;
    nxtfree = (nxtfree+1)%SIZE;
    V(full);
  }
}
process Consumer {
  int data;
  while (true) {
    P(full);
    data = buf[nxtdata];
    nxtdata = (nxtdata+1)%SIZE;
    V(empty);
    /* consome dados */
  }
}
```

Buffer Limitado - n processos de cada tipo

```
int buf[SIZE]; int nxtfree = 0; int nxtdata = 0;
-- uso de semáforos contadores e de exclusão mútua
sem empty = SIZE; sem full = 0; sem exc = 1;
process Producer {
  int data;
  while (true) {
    /* produz dados */
    P(empty); P(exc);
    buf[nxtfree] = data;
    nxtfree = (nxtfree+1)%SIZE;
    V(exc); V(full);
  }
}
process Consumer {
  int data;
  while (true) {
    P(full); P(exc);
    data = buf[nxtdata];
    nxtdata = (nxtdata+1)%SIZE;
    V(exc); V(empty);
    /* consome dados */
  }
}
```

Outros exemplos

- barreira
- filósofos

- estudo de padrões de problemas

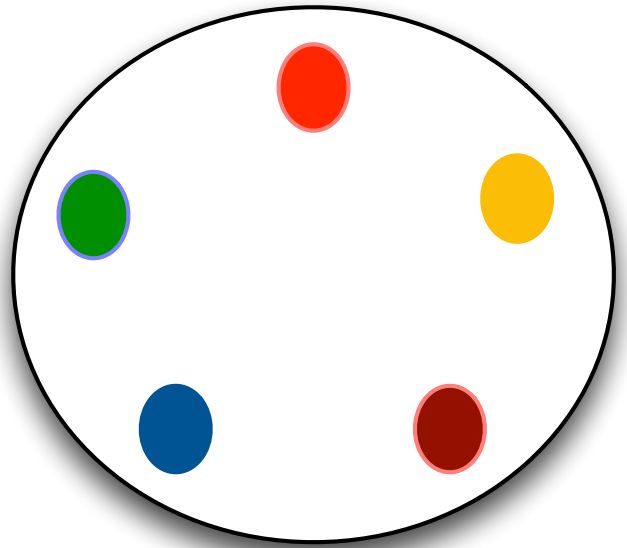
Barreira

- threads só podem prosseguir quando todos tiverem chegado ao ponto da "barreira"

Barreira

Filósofos

- conjuntos de recursos com interseção



Filósofos


Leitores e Escritores

- escritores precisam de acesso exclusivo ao recurso
- leitores podem acessar o recurso concorrentemente com outros leitores

- solução clássica: contamos quantos leitores/escritores estão acessando a base

E fora desses padrões...?

- em alguns casos o desenho de uma solução por semáforos é bastante complicado
  - leitores e escritores

Passagem de Bastão

- em alguns casos o desenho de uma solução por semáforos é bastante complicado
- exemplo: problema de leitores e escritores

A técnica de passagem de bastão

Criamos:

- um semaforo e associado à entrada nos comandos atômicos ($< await(B)S; >$ ou $< S :>$)
- um semáforo `sb` e um contador `db` para cada condição B distinta (inicializados com 0)

Cada $< await(B)S; >$ fica associado ao seguinte código:

```
P(e)
if (!B) { db++; V(e); P(sb);}
S;
SIGNAL
```

onde o trecho "SIGNAL" corresponde à passagem de bastão para outros processos que estejam em espera

Voltando ao Buffer

```
int buf[SIZE]; int nxtfree = 0; int nxtdata = 0;
process Producer {
  int data;
  while (true) {
    /* produz dados */
    < await ((nxtfree+1)%SIZE != nxtdata)
    buf[nxtfree] = data;
    nxtfree = (nxtfree+1)%SIZE; >
  }
}
process Consumer {
  int data;
  while (true) {
    < await ((nxtfree != nxtdata)
    data = buf[nxtdata];
    nxtdata = (nxtdata+1)%SIZE; >
    /* consome dados */
  }
}
```


Solução com Passagem de Bastão para Prod/Cons

- mais complicada que a "ad-hoc"
- ... porém derivada de forma metódica...

Problema da Barreira

- ponto de sincronização: processos devem esperar todos os demais chegarem a determinado ponto de execução

```
int chegaram[NUMPROCS] = [0, .., 0];

process trab [i = 1 to n] {
  ...
  for (iter = 0; iter < NUMITERS; iter++) {
    trabalha;
    barreira(iter);
  }
}
void barreira (int iter) {
  <chegaram[iter]++>
  <await chegaram[iter] == NUMPROCS>
}
```

Leitores e Escritores

```
int nr = 0, nw = 0;
## RW:  (nr == 0 ∨ nw == 0) ∧ nw <= 1
process Reader[i = 1 to m] {
  while (true) {
    ...
    ⟨await (nw == 0) nr = nr+1;⟩
    read the database;
    ⟨nr = nr-1;⟩
  }
}
process Writer[j = 1 to n] {
  while (true) {
    ...
    ⟨await (nr == 0 and nw == 0) nw = nw+1;⟩
    write the database;
    ⟨nw = nw-1;⟩
  }
}
```

- especificação baseada em $< await >$

Problema dos leitores e escritores

- figuras Andrews

exercício 1

Implemente uma estrutura de dados que é um buffer limitado com N posições, usado para *broadcast* entre P produtores e C consumidores. O buffer oferece operações `deposita` e `consome`. Ao chamar `deposita`, o produtor deve ficar bloqueado até conseguir inserir o novo item, e ao chamar `consome` o consumidor deve ficar bloqueado até conseguir um item para consumir. Uma posição só pode ser reutilizada quando todos os C consumidores tiverem lido a mensagem. Cada consumidor deve receber as mensagens na ordem em que foram depositadas. Programar em C (ou C++) com passagem de bastão.