Programação Concorrente – Aula 6

Gilmar Gimenes Rodrigues

1 Solução do Problema da Região Crítica Usando `await`

1.1 Para duas threads:

```
boolean in1 = false;
        in2 = false;

thread_1() {
    while (true) {
        <await !in2; in = true;>   /* protocolo de entrada */
        regiao_critica();
        in1 = false;               /* protocolo de saida */
        regiao_nao_critica();
    }
}

thread_2() {
    while(true) {
        <await !in1; in2 = true;>
        regiao_critica();
        in2 = false;
        regiao_nao_critica();
    }
}
```

1.2 Para n threads:

```
boolean ocupado = false;

thread_i() {
    while (true) {
        <await !ocupado; ocupado = true;>
        regiao_critica();
        ocupado = false;
        região_nao_critica();
    }
}
```

Mesma solução, escrita de outra forma (só os protocolos de entrada e saída):

```
boolean ocupado = false;

entra_regiao_critica() {
    <await !ocupado; ocupado = true;>
}

sai_regiao_critica() {
    ocupado = false;
}
```

Como implementar esse `await`?

Com a ajuda do hardware: instruções "test and set" ou "swap", incremento atômico, fetch and add, ldots

2 Test and Set

Instrução atômica com 2 operandos:

- Uma variável `cond` compartilhada entre threads.
- Uma variável `result` não compartilhada, local à thread. Tipicamente, `result` é um registrador ou flag da CPU.

Por definição, `test_and_set(cond, result)` é `<result = cond; cond = true>`.

Obs.: A instrução `test_and_set(cond, result)` faz dois acessos à memória (leitura e escrita de `cond`).

Outra instrução atômica: `swap`, com os 2 operandos abaixo.

- Variável `c`: compartilhada entre threads
- Variável `l`: não compartilhada (tipicamente é registrador da CPU)

Por definição, `swap(c, l)` é `<temp = c; c = l; l = temp>`.

Tendo `swap`, pode-se obter o efeito de `test_and_set`:

```
result = true;
swap(cond, result);
```

Essas duas linhas são é equivalentes a um `test_and_set(cond, result)`.

O x86 tem uma instrução do tipo "swap", mas com o mnemônico `XCHG`:

```
XCHG    mem, reg        ; troca o valor do registrador com a memória.

MOV     AX, 1           ; estas duas instruções
XCHG    ocupado, AX     ; fazem um test and set
```

Solução do problema da região crítica usando "test and set":

```
boolean ocupado = false;

entra_regiao_critica() {
    boolean valor_lido;
    test_and_set(ocupado, valor_lido);
    while (valor_lido) {
        test_and_set(ocupado, valor_lido);  /* busy wait */
    }
}

sai_regiao_critica() {
    ocupado = false;
}
```

A mesma solução em linguagem de montagem:

```
entra_RC:
        MOV     AX, 1
        XCHG    ocupado, AX
        JNZ     entra_RC
        ret

sai_RC:
        MOV     ocupado, 0
        ret
```

Desvantagens dessa solução:

1. Não garante justiça.
2. Memory contention

   Várias threads terão acesso à variável `ocupado`, mas apenas uma por vez.
3. Invalidação de cache em sistemas MP (multi-processadores)

   As threads escrevem `true` na variável `ocupado` mesmo quando essa variável já contém `true`.

Para diminuir o problema de invalidação do cache:

```
entra_regiao_critica() {
    boolean valor_lido;
    while (ocupado)
        ;
    test_and_set(ocupado, valor_lido)
    while (valor_lido) {
        while (ocupado) {
            test_and_set(ocupado, valor_lido);
        }
    }
}
```

3 Implementação de Ações Atômicas Coarse-Grained

3.1 Ação Atômica Incondicional

A ação atômica composta

```
<comando_1; comando_2; ...; comando_n;>
```

pode ser implementada como

```
entra_regiao_critica();
comando_1;
comando_2;
  .
  .
  .
comando_n;
sai_regiao_critica();
```

3.2 Ação Atômica Condicional

A ação atômica com `await`

```
<await c; cmd_1; cmd_2; ...; cmd_n;>
```

pode ser implementada como

```
entra_regiao_critica();
while (!c) {
    sai_regiao_critica();
    delay(); /* dá chance para outras threads entrarem
                na rc e tornarem a condição c verdadeira */
    entra_regiao_critica();
}
cmd_1; cmd_2; ...; cmd_n;
sai_regiao_critica();
```

4 Algoritmo do Ticket

Usando ações atômicas coarse-grained:

```
int proximo; // o que está sendo chamado para atendimento
int numero;  // número do proximo ticket distribuido

entra_regiao_critica() {
    int minha_vez;
    <minha_vez = numero; numero = numero + 1>
    <await proximo == minha_vez;>
}

sai_regiao_critica() {
    proximo = proximo + 1; /* No máximo uma thread entra na r.c., portanto
                              só uma thread sairá dela. Logo esta instrução
                              não precisa ser atômica. */
}
```

Algoritmo do ticket sem ações coarse-grained, usando a instrução "fetch and add":

```
int proximo; // o que está sendo chamado para atendimento
int numero;  // número do proximo ticket distribuido

entra_regiao_critica() {
    int minha_vez;
    minha_vez = fetch_and_add(numero, 1);
    while (proximo != minha_vez)
        ;
}

sai_regiao_critica () {
    proximo = proximo + 1; /* No máximo uma thread entra na r.c., portanto
                              só uma thread sairá dela. Logo esta instrução
                              não precisa ser atômica. */
}
```

A instrução atômica `fetch_and_add(var, incr)` é definida como

```
<temp = var; var = var + incr; return temp> .
```

Infelizmente ela é disponibilizada por poucas máquinas (o x86 não tem `fetch_and_add`).