Programação Concorrente – Aula 3

Gilmar Gimenes Rodrigues

1 O Problema da Região Crítica

Requisitos:

- exclusão mútua
- ausência de deadlock
- ausência de atraso desnecessário
- garantia de entrada

Hipótese: threads não podem morrer dentro de um região crítica.

"Solução" em sistemas com uma só CPU: desabilitar interrupções. Precisamos de soluções melhores!

2 Outra Solução: Alternância Estrita

```
int vez = 0; /* 0 ou 1, indicando qual thread tem a "vez" */

thread_0 () {
    while (true) {
        while (vez != 0) /* protocolo de entrada (espera a vez) */
            ;
        região_critica();
        vez = 1; /* protocolo de saida (passa a vez) */
        regiao_nao_critica();
    }
}

thread_1 () {
    while (true) {
        while (vez != 1) /* protocolo de entrada (espera a vez) */
            ;
        região_critica();
        vez = 0 /* protocolo de saida (passa a vez) */
        regiao_nao_critica();
    }
}
```

- Tem exclusão mútua.
- Tem ausência de deadlock.

- Mas nao é solução porque:
  - Possui atrasos desnecessários.
  - Não possui garantia de entrada:

    Se uma das threads morrer fora da região crítica a outra thread não conseguirá mais entrar na região crítica.
- Sem ajuda de hardware.
- Fácil de generalizar para "n" threads.

Ainda alternância estrita:

O mesmo algoritmo, escrito de outra forma (só os protocolos de entrada e de saída da r.c.):

```
vez = 0;  /* 0 ou 1 */

entra_regiao_critica(int i) {  /* i é o número da thread (0 ou 1) */
    while (vez != i)
        ;
}

sai_regiao_critica(int i) {
    vez = 1 - i; /* a outra thread */
}
```

3 Algoritmo de Dekker para Duas Threads

Idéia: usar um critério de alternância só se as duas threads estiverem querendo entrar na região crítica. Se só uma thread quer entrar na r.c., ela não precisa esperar nada. Se as duas threads querem entrar ao mesmo tempo, uma delas (usando um critério de alternância) deixa a outra ir primeiro.

```
boolean quer_entrar[2] = {false, false};
int vez = 0;  /* para alternar a vez */

entra_regiao_critica(int i) { /* i é o número da thread (0 ou 1) */
    int outra = 1 - i;
    quer_entrar[i] = true;
    while (quer_entrar[outra]) {
        if (vez == outra) {
            quer_entrar[i] = false;
            while (vez == outra)
                ;
            quer_entrar[i] == true;
        }
    }
}

sair_regiao_critica (int i) { /* i é o número da thread (0 ou 1) */
    quer_entrar[i] = false;
    vez = 1 - i; /* outra */
}
```

4 Algoritmo de Peterson (Tie Breaker) para Duas Threads

Primeira tentativa:

```
boolean quer_entrar[2] = {false, false};

entra_regiao_critica (int i) {
    int outro = 1 - i;
    while (quer_entrar[outro])
        ;           /* espera a outra thread sair da r.c. */
    quer_entrar[i] = true;
}

sai_regiao_critica(int i) {
    quer_entrar[i] = false;
}
```

Não funciona porque as duas threads podem entrar na região crítica ao mesmo tempo.

Segunda tentativa:

```
entra_regiao_critica (int i) {
    int outro = 1 - i;
    quer_entrar[i] = true;
    while (quer_entrar[outro])
        ;           /* espera a outra thread sair da r.c. */
}
```

Ocorre deadlock!

Algoritmo de Peterson:

```
boolean quer_entrar[2] = {false, false};
int ultimo;  /* ultimo a tentar */

entra_regiao_critica(int i) {
    int outra = 1 - i;
    quer_entrar[i] = true;
    ultimo = i;
    while (quer_entrar[outro] && ultimo == i)
        ;           /* espera ocupada */
}

sai_regiao_critica(int i) {
    quer_entrar[i] = false;
}
```

Muito mais simples que o algoritmo de Dekker! Só uma crítica: em vez da espera ocupada, seria preferível que a CPU fosse fazer outra coisa (alguma tarefa útil).