

MC504/MC514 - Sistemas Operacionais

Processos e Threads: Exclusão mútua

Islene Calciolari Garcia

Primeiro Semestre de 2016

Sumário

- 1 Condição de corrida
- 2 Exclusão mútua
- 3 Desempate para N threads
- 4 Thread Gerente

Acesso a recursos compartilhados

- Estudo de caso:

```
volatile int s; /* Variável compartilhada */
```

```
/* Cada thread tentar executar os seguintes  
   comandos sem interferência.  */
```

```
s = thr_id;  
printf ("Thr %d: %d", thr_id, s);
```

Condição de corrida

Saída esperada

```
volatile int s; /* Variável compartilhada */
```

Thread 0

- (i) `s = 0;`
- (ii) `print ("Thr 0: ", s);`

Thread 1

- (iii) `s = 1;`
- (iv) `print ("Thr 1: ", s);`

Saída: Thr 0: 0
 Thr 1: 1

Condição de corrida

Saída esperada II

```
volatile int s; /* Variável compartilhada */
```

Thread 0

- (iii) `s = 0;`
- (iv) `print ("Thr 0: ", s);`

Thread 1

- (i) `s = 1;`
- (ii) `print ("Thr 1: ", s);`

Saída: Thr 1: 1
 Thr 0: 0

Condição de corrida

Saída inesperada

```
volatile int s; /* Variável compartilhada */
```

Thread 0

(i) `s = 0;`

(iii) `print ("Thr 0: ", s);`

Thread 1

(ii) `s = 1;`

(iv) `print ("Thr 1: ", s);`

Saída: Thr 0: 1

Thr 1: 1

Veja o código: `inesperada.c`

Exclusão mútua

- Acesso controlado a recursos compartilhados
- Estudo de caso:

```
volatile int s; /* Variável compartilhada */
while (1) {
    /* Região não crítica */
    /* Protocolo de entrada */
    /* Região crítica */
    s = thr_id;
    printf ("Thr %d: %d", thr_id, s);
    /* Protocolo de saída */
}
```

Exclusão Mútua

- Os algoritmo devem garantir:
 - exclusão mútua
 - ausência de *deadlock*
 - ausência de *starvation*
 - progresso (uma thread que não esteja interessada na região crítica não pode impedir outra thread de entrar na região crítica)

Tentando implementar um lock

- Lock = variável compartilhada com o seguinte significado:
 - `lock == 0` \Rightarrow região crítica está livre
 - `lock != 0` \Rightarrow região crítica está ocupada
- Protocolo de entrada na região crítica

```
while (lock != 0);
```
- Protocolo de saída da região crítica

```
lock = 0;
```

Tentando implementar um lock

```
volatile int s = 0, lock = 0;
```

Thread 0

```
while (lock == 1);  
lock = 1;  
s = 0;  
print ("Thr 0:" , s);  
lock = 0;
```

- Veja o código: tentativa_lock.c

Thread 1

```
while (lock == 1);  
lock = 1;  
s = 1;  
print ("Thr 1:" , s);  
lock = 0;
```

Solução em hardware

entra_RC:

TSL RX, lock

CMP RX, #0

JNE entra_RC

RET

deixa_RC:

MOV lock, \#0

RET

- Instrução *test and set* executa atomicamente:
 - lê o conteúdo da variável lock;
 - armazena este conteúdo em RX;
 - coloca um valor não nulo em lock.
- Não vale para a aula de hoje :-)

Solução em hardware

entra_RC:

```
MOV lock,#1
XCHG RX, lock
CMP RX, #0
JNE entra_RC
RET
```

deixa_RC:

```
MOV lock, #0
RET
```

- Instrução *exchange* troca atonicamente os conteúdos do registrador e da memória;
- Também não vale para a aula de hoje :-)

Abordagem da Alternância

```
int s = 0;  
int vez = 1; /* Primeiro a thread 1 */
```

Thread 0

```
while (true)  
    while (vez != 0);  
    s = 0;  
    print ("Thr 0:" , s);  
    vez = 1;
```

- Veja o código: `alternancia.c`

Thread 1

```
while (true)  
    while (vez != 1);  
    s = 1;  
    print ("Thr 1:" , s);  
    vez = 0;
```

Abordagem da Alternância

N threads

Thread_i:

```
while (true)
    while (vez != i);
    s = i;
    print ("Thr ", i, ": ", s);
    vez = (i + 1) % N;
```

- Veja o código: `alternanciaN.c`

Limitações da Alternância

- Uma thread fora da RC pode impedir outra thread de entrar na RC
- Se uma thread interromper o ciclo a outra não poderá mais entrar na RC

Vetor de Interesse

```
int s = 0;  
int interesse[2] = {false, false};
```

Thread 0

```
while (true)  
    interesse[0] = true;  
    while (interesse[1]);  
    s = 0;  
    print("Thr 0:" , s);  
    interesse[0] = false;
```

- Veja o código: interesse.c

Thread 1

```
while (true)  
    interesse[1] = true;  
    while (interesse[0]);  
    s = 1;  
    print("Thr 1:" , s);  
    interesse[1] = false;
```


Limitações do Vetor de Interesse

- O algoritmo anterior garante exclusão mútua, mas...
- se as duas threads ficarem interessadas ao mesmo tempo haverá *deadlock*.
- Podemos tentar sanar este problema da seguinte forma:
Se as duas threads ficarem interessadas ao mesmo tempo, elas irão baixar o interesse, esperar um pouco e tentar novamente.
- Veja o código: `interesse2.c`

Vetor de Interesse II

```
int s = 0;  
int interesse[2] = {false, false};
```

Thread 0

```
while (true)  
    interesse[0] = true;  
    while (interesse[1])  
        interesse[0] = false;  
        sleep(1);  
        interesse[0] = true;  
s = 0;  
print("Thr 0:" , s);  
interesse[0] = false;
```

Thread 1

```
while (true)  
    interesse[1] = true;  
    while (interesse[0])  
        interesse[1] = false;  
        sleep(1);  
        interesse[1] = true;  
s = 1;  
print("Thr 1:" , s);  
interesse[1] = false;
```

Limitações do Vetor de Interesse II

- O algoritmo anterior garante exclusão mútua, mas...
- se as duas threads andarem sempre no mesmo passo haverá *livelock*.
- Podemos tentar outra abordagem que é:
Se as duas threads ficarem interessadas ao mesmo tempo, entrará na região crítica a thread cujo identificador estiver marcado na variável vez.
- Veja o código: `interesse_vez.c`

Vetor de Interesse e Alternância

```
int s = 0, vez = 0;  
int interesse[2] = {false, false};
```

Thread 0

```
while (true)  
    interesse[0] = true;  
    if (interesse[1])  
        while (vez != 0);  
    s = 0;  
    print("Thr 0:", s);  
    vez = 1;  
    interesse[0] = false;
```

Thread 1

```
while (true)  
    interesse[1] = true;  
    if (interesse[0])  
        while (vez != 1);  
    s = 1;  
    print("Thr 1:", s);  
    vez = 0;  
    interesse[1] = false;
```

Limitações da combinação anterior

- O algoritmo anterior não garante exclusão mútua. Você consegue indicar um cenário?
- Podemos tentar melhorar o algoritmo:
Se as duas threads ficarem interessadas ao mesmo tempo, elas deverão baixar o interesse e esperar por sua vez.
- Veja o código: `quase_dekker.c`

Quase o algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
    interesse[0] = true;
    while (interesse[1])
        interesse[0] = false;
        while (vez != 0);
        interesse[0] = true;
    s = 0;
    print ("Thr 0:" , s);
    vez = 1;
    interesse[0] = false;
```

Thread 1

```
while (true)
    interesse[1] = true;
    while(interesse[0])
        interesse[1] = false;
        while(vez != 1);
        interesse[1] = true;
    s = 1;
    print ("Thr 1:" , s);
    vez = 0;
    interesse[1] = false;
```

Limitações do algoritmo anterior

- O algoritmo anterior garante exclusão mútua?
- É possível que uma thread ganhe sempre a região crítica enquanto a outra fica só esperando?
- Podemos melhorar o algoritmo:
Se as duas threads ficarem interessadas ao mesmo tempo, a thread da vez não baixa o interesse.
- Veja o código: `dekker.c`

Algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
    interesse[0] = true;
    while (interesse[1])
        if (vez != 0)
            interesse[0] = false;
            while (vez != 0);
            interesse[0] = true;
    s = 0;
    print ("Thr 0:" , s);
    vez = 1;
    interesse[0] = false;
```

Thread 1

```
while (true)
    interesse[1] = true;
    while(interesse[0])
        if (vez != 1)
            interesse[1] = false;
            while(vez != 1);
            interesse[1] = true;
    s = 1;
    print ("Thr 1:" , s);
    vez = 0;
    interesse[1] = false;
```


Algoritmo do Desempate (1981)

```
int s = 0, ultimo = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
    interesse[0] = true;
    ultimo = 0;
    while (ultimo == 0 &&
           interesse[1]);
    s = 0;
    print ("Thr 0:" , s);
    interesse[0] = false;
```

Thread 1

```
while (true)
    interesse[1] = true;
    ultimo = 1;
    while (ultimo == 1 &&
           interesse[0]);
    s = 1;
    print ("Thr 1:" , s);
    interesse[1] = false;
```

Abordagem da Alternância

N threads

Thread_i:

```
while (true)
    while (vez != i);
    s = i;
    print ("Thr ", i, ": ", s);
    vez = (i + 1) % N;
```

- Veja o código: `alternanciaN.c`

Vetor de interesse

N threads

```
int interesse[N] = {false, ..., false}
while (true) { /* Código da Thread_i */
    interesse[i] = true;
    while (existe j!=i tal que (interesse[j]));
    s = i;
    print ("Thr ", i, ": ", s);
    interesse[i] = false;
```

- Veja o código: interesseN.c

Algoritmo do Desempate (1981)

```
int s = 0, ultimo = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
    interesse[0] = true;
    ultimo = 0;
    while (ultimo == 0 &&
           interesse[1]);
    s = 0;
    print ("Thr 0:" , s);
    interesse[0] = false;
```

Thread 1

```
while (true)
    interesse[1] = true;
    ultimo = 1;
    while (ultimo == 1 &&
           interesse[0]);
    s = 1;
    print ("Thr 1:" , s);
    interesse[1] = false;
```

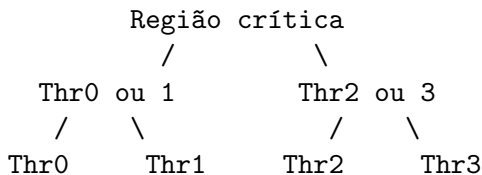
Algoritmo do Desempate

Características

Região crítica
/ \
Thr0 Thr1

- Funciona para 2 threads
- Variável ultimo é acessada pelas 2 threads
- Variável interesse[i] é acessada
 - para escrita pela thread i
 - para leitura pela thread adversária

Campeonato entre 4 threads



- A thread campeã da disputa entre Thr0 e Thr1 disputa a região crítica com a thread campeã da disputa entre Thr2 e Thr3.
- Todas as partidas são instâncias do algoritmo do desempate.

Campeonato entre 4 threads

Variáveis de controle replicadas

```
int ultimo_final = 0;  
int interesse_final[2] = {false, false};
```

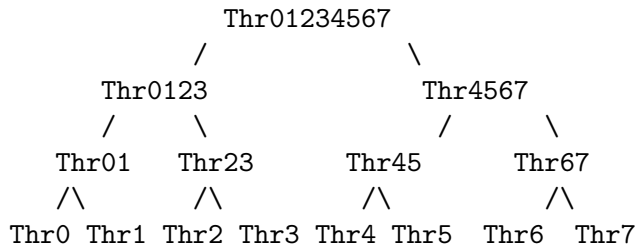
```
int ultimo01 = 0;  
int interesse01[2] = {false, false};
```

```
int ultimo23 = 2;  
int interesse23[2] = {false, false};
```

- Veja código: camp4.c
- Como implementar com uma única função para todas as threads?

Exclusão mútua entre N threads

Abordagem do campeonato



- As threads podem concorrer duas a duas
- Garante ausência de starvation?

Thread Gerente

- E se tivéssemos uma thread gerente?
 - Os algoritmos seriam mais simples?
 - Qual o grande ponto negativo desta abordagem?
- Veja os programas: gerente?.c