MO806/MC914
Tópicos em Sistemas Operacionais
2s2006

Processos e Threads 3

Algoritmos de Exclusão Mútua

- Devemos garantir:
  - exclusão mútua
  - ausência de deadlock
  - progresso (uma thread que não esteja interessada na região crítica não pode impedir outra thread de entrar na região crítica)
- Como escrever provas formais?

Vetor de Interesse

```
int i[2] = {false, false};
```

Thread 0

```
while (true)
  a0:  nao_critica();
  b0:  i[0] = true;
  c0:  while (i[1]);
  d0:  critica();
  e0:  i[0] = false;
```

Thread 1

```
while (true)
  a1:  nao_critica();
  b1:  i[1] = true;
  c1:  while (i[0]);
  d1:  critica();
  e1:  i[1] = false;
```

- Veja o código: interesse.c

Prova - exclusão mútua

$$i[0] \equiv at(c0) \vee at(d0) \vee at(e0)$$

$$i[1] \equiv at(c1) \vee at(d1) \vee at(e1)$$

$$\neg(at(d0) \wedge at(d1))$$

Fonte: Principles of Concurrent and Distributed Programming - M. Ben-Ari

Prova - exclusão mútua

$$i[0] \equiv at(c0) \vee at(d0) \vee at(e0)$$

- A fórmula é inicialmente válida
- $a0 \rightarrow b0$ - não altera a fórmula
- $b0 \rightarrow c0$ - altera os dois lados da fórmula
- $c0 \rightarrow c0$, $c0 \rightarrow d0$ e $d0 \rightarrow e0$ - não alteram a validade de nenhuma dos dois lados da fórmula
- $e0 \rightarrow a0$ - altera os dois lados da fórmula
- Transições na thread 1 não alteram a fórmula

Prova - exclusão mútua

$$\neg(at(d0) \wedge at(d1))$$

- A fórmula é inicialmente válida
- Considere $at(d0)$ e que a thread 1 vai fazer a transição $c1 \rightarrow d1$
- $at(d0)$ implica $i[0]$ e, portanto, a thread 1 fica presa no loop

- Cenários simétricos $\Rightarrow$ provas similares

Algoritmo de Dekker

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

Thread 0

```
while (true)
  a0:  nao_critica();
  b0:  i[0] = true;
  c0:  while (i[1])
    d0:  if (vez != 0)
      e0:  i[0] = false;
      f0:  while (vez !=0);
      g0:  i[0] = true;
  h0:  critica();
  i0:  vez = 1;
  j0:  i[0] = false;
```

Thread 1

```
while (true)
  a1:  nao_critica();
  b1:  i[1] = true;
  c1:  while(i[0])
    d1:  if (vez != 1)
      e1:  i[1] = false;
      f1:  while(vez != 1);
      g1:  i[1] = true;
  h1:  critica();
  i1:  vez = 0;
  j1:  i[1] = false;
```

Prova - exclusão mútua

$$i[0] \equiv at(c0) \vee at(d0) \vee at(e0) \vee at(h0) \vee at(i0) \vee at(j0)$$

$$i[1] \equiv at(c1) \vee at(d1) \vee at(e1) \vee at(h1) \vee at(i1) \vee at(j1)$$

$$\neg(at(h0) \wedge at(h1))$$

Prova - ausência de starvation

Lógica Temporal

$\Box$: *definitely* $\quad$ $\Diamond$: *eventually*

$$\Box i[0] \wedge \Box vez = 0 \supset \Diamond\Box i[1] = 0$$

$$\Box i[0] \wedge \Box vez = 1 \supset \quad \Diamond\, vez = 0$$

Sugestão para N threads

```
int vez = 0, interesse = {false, ..., false}
while (true) {  /* Código da Thread_i */
   interesse[i] = true;
   while (existe j!=i tal que (interesse[j]))
     if (vez != i)
       interesse[i] = false;
       while (vez != i) ;
       interesse[i] = true;
    s = i;
    print ("Thr ", i, ": ", s);
    vez = (i+1) % N;
    interesse[i] = false;
```

Sugestão para N threads

Garante exclusão mútua?

- Uma thread só entra na região crítica após percorrer o vetor e verificar que nenhuma outra está interessada.

Garante ausência de deadlock?

- Se todas estiverem interessadas, pelo menos uma thread (a da vez) sempre consegue entrar na região crítica

Garante progresso sempre?

- Não. A vez pode ser passada para uma thread desinteressada.
- Veja o código dekkerN.c

Outra sugestão...

```
int vez = 0, interesse = {false, ..., false}
while (true) {  /* Código da Thread_i */
   interesse[i] = true;
   while (existe j!=i tal que (interesse[j]))
     if (vez == -1 || vez != i)
       interesse[i] = false;
       while (vez == -1 || vez != i) ;
       interesse[i] = true;
```

Outra sugestão...

```
s = i;
print ("Thr ", i, ": ", s);
vez = alguma interessada ou -1;
interesse[i] = false;
```

Por que não funciona?

- Por que mais de uma thread pode achar que é a vez dela ao encontrar vez == -1
- Veja o código: outro_dekker.N
- Você tem outra sugestão?

Algoritmo do Desempate

2 Threads

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

Thread 0

```
while (true)
  a0:  nao_critica();
  b0:  i[0] = true;
  c0:  vez = 0;
  d0:  while (vez == 0
           && i[1]);
  e0:  critica();
  f0:  i[0] = false;
```

Thread 1

```
while (true)
  a1:  nao_critica();
  b1:  i[1] = true;
  c1:  vez = 1;
  d1:  while (vez == 1
           && i[0]);
  e1:  critica();
  f1:  i[1] = false;
```

Algoritmo do Desempate

Invariantes

$$\Box\neg i[1] \vee \Diamond(vez == 1)$$

$$at(d0) \wedge \Diamond\neg at(e0) \supset \Box\Diamond i[1]$$

$$at(d0) \wedge \Diamond\neg at(e0) \supset \Diamond(vez == 1)$$

Algoritmo do Desempate

3 Threads (bug!)

```
int s=0, vez=0, interesse[3] = {false,false,false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  vez = 0;
  while (vez == 0 && (interesse[1] || interesse[2]));
  s = 0;
  print ("Thr 0:" , s);
  interesse[0] = false;
```

Algoritmo do desempate

Extensão para 3 threads

- Para 2 threads, podemos estabelecer que a thread que alterou vez por último perde;
- Caso 3 threads alterem a variável vez simultaneamente, só poderemos identificar a que fez a última alteração.
- Como indicar que 2 threads perderam?

Algoritmo do Desempate

3 Threads (bug?)

```
int s=0, vez0, vez1, interesse[3] = {false,false,false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  vez0 = 0;
  while (vez0 == 0 && interesse[1] && interesse[2]);
  vez1 = 0;
  while (vez1 == 0 && (interesse[1] || interesse[2]));
  s = 0;
  print ("Thr 0:" , s);
  interesse[0] = false;
```

Algoritmo do desempate

Extensão para N threads

- Caso M threads alterem a variável vez simultaneamente, só poderemos identificar a que fez a última alteração.
- Como indicar que $M - 1$ threads perderam?

Algoritmo do desempate

N threads

- Dividimos o problema em N-1 fases (0..N-2)
- A cada fase, conseguimos identificar uma thread perdedora, que fica esperando
- Variáveis de controle:

```
int interesse[N];
int fase[N];
int vez[N-1];
```

Desempate para N threads

Estado inicial

	Thr0	Thr1	Thr2	Thr3	Thr4
interesse	false	false	false	false	false
fase	-1	-1	-1	-1	-1

	Fase0	Fase1	Fase2	Fase3
vez	–	–	–	–

Desempate para N threads

Todas as threads interessadas

	Thr0	Thr1	Thr2	Thr3	Thr4
interesse	true	true	true	true	true
fase	0	0	0	0	0

	Fase0	Fase1	Fase2	Fase3
vez	2	–	–	–

- Thread 2 não poderá mudar de fase

Desempate para N threads

Todas as threads interessadas

	Thr0	Thr1	Thr2	Thr3	Thr4
interesse	true	true	true	true	true
fase	1	1	0	1	1

	Fase0	Fase1	Fase2	Fase3
vez	2	1	–	–

- Thread 1 não poderá mudar de fase

Desempate para N threads

Todas as threads interessadas

	Thr0	Thr1	Thr2	Thr3	Thr4
interesse	true	true	true	true	true
fase	2	1	0	2	2

	Fase0	Fase1	Fase2	Fase3
vez	2	1	0	–

- Thread 0 não poderá mudar de fase

Desempate para N threads

Todas as threads interessadas

	Thr0	Thr1	Thr2	Thr3	Thr4
interesse	true	true	true	true	true
fase	2	1	0	3	3

	Fase0	Fase1	Fase2	Fase3
vez	2	1	0	4

- Thread 3 pode entrar na região crítica

Desempate para N threads

Algumas threads interessadas

	Thr0	Thr1	Thr2	Thr3	Thr4
interesse	true	true	false	false	false
fase	2	0	-1	-1	-1

	Fase0	Fase1	Fase2	Fase3
vez	1	0	0	–

- Thread 1 deverá esperar

Desempate para N threads

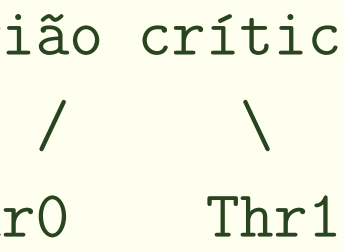
```
int interesse[N], fase[N], vez[N-1];
```

Thread_i:

```
   interesse[i] = true;
   for (f = 0; f < N-1; f++)
      fase[i] = f;
      vez[f] = i;
      for (j = 0; j < N && vez[f] == i; j++ )
         if (j != i && interesse[j])
            while (f <= fase[j] && vez[f] == i);
   s = i;
   print ("Thr ", i, s);
   interesse[i] = false;
   fase[i] = -1;
```

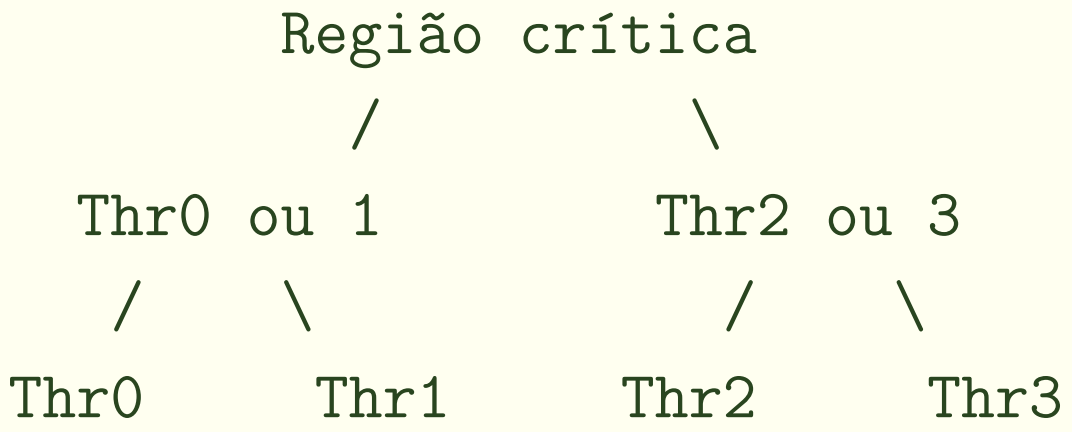
Algoritmo do Desempate

Características



- Funciona para 2 threads
- Variável vez 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

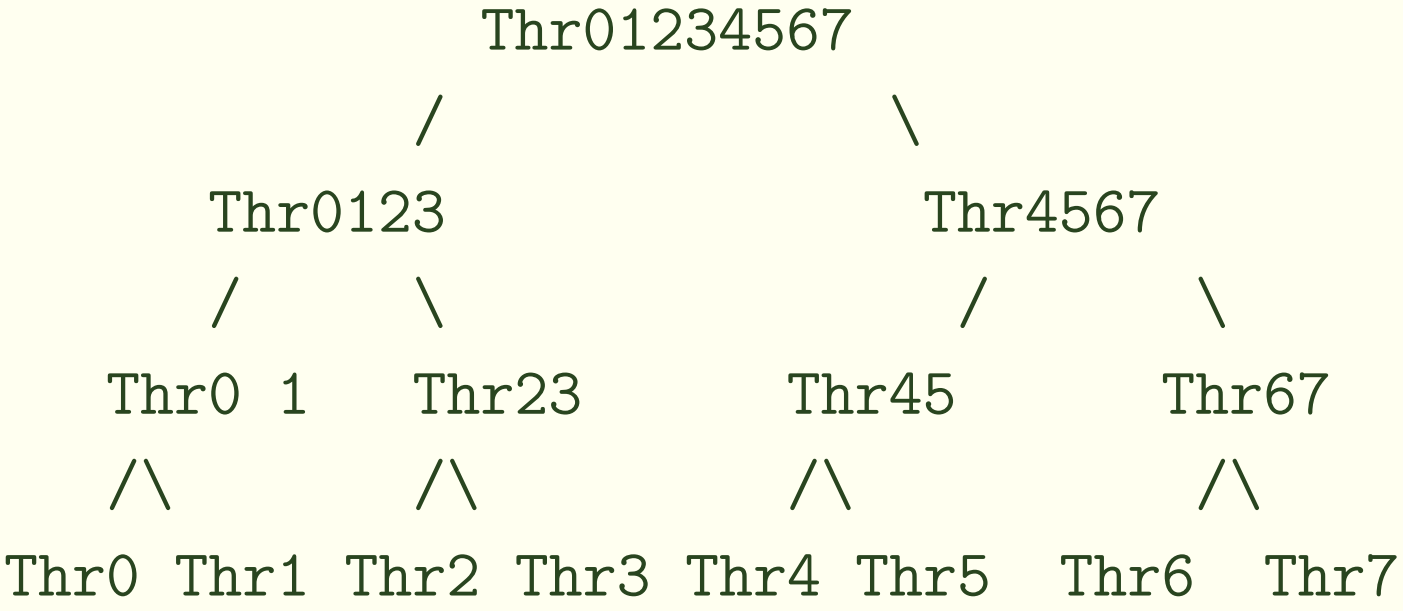
Variáveis de controle replicadas

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

int vez01 = 0;
int interesse01[2] = {false, false};

int vez23 = 2;
int interesse23[2] = {false, false};
```

- Veja código: camp4.c

Exclusão mútua entre N threads

Abordagem do campeonato



- As threads podem concorrer duas a duas
- Pode ser aplicada a qualquer algoritmo