MC514–Sistemas Operacionais: Teoria e Prática

1s2010

Processos e Threads 3

Objetivos

- Primeiros problemas de condição de corrida
- Exclusão mútua
- Primeiras tentativas de algoritmos
- Algoritmo de Dekker
- Algoritmo do desempate

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 disputa

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 disputa

Saída esperada II

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

	Thread 0		**Thread 1**
(iii)	`s = 0;`	(i)	`s = 1;`
(iv)	`print ("Thr 0:  ", s);`	(ii)	`print ("Thr 1:  ", s);`

Saída: Thr 1: 1

Thr 0: 0

Condição de disputa

Saída inesperada

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

	Thread 0		**Thread 1**
(i)	`s = 0;`	(ii)	`s = 1;`
(iii)	`print ("Thr 0:  ", s);`	(iv)	`print ("Thr 1:  ", s);`

Saída: Thr 0: 1

Thr 1: 1

Veja o código: `inesperada.c`

Escalonamento de threads

- A execução de uma thread pode ser interrompida a qualquer momento.
- Veja o código `preemptivo.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 */
  }
```

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;
```

Thread 1

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

- Veja o código: tentativa_lock.c

Solução em hardware

```
entra_RC:
    TSL RX, lock
    CMP RX, #0
    JNE entra_RC
    RET

deixa_RC:
    MOV lock, \#0
    RET
```

- 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;
```

Thread 1

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

- Veja o código: alternancia.c

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;
```

Thread 1

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

- Veja o código: interesse.c

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?

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

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;
```

Prova - deadlock

- Basta apresentar um escalonamento: $a0\ a1\ b0\ b1$

Prova - exclusão mútua

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

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

$$\text{Exclusão mútua} \equiv \neg(at(d0) \wedge at(d1))$$

Prova - exclusão mútua

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

- $a0$ 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 nenhum 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 e não consegue completar a transição
- Cenários simétricos $\Rightarrow$ provas similares

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;
```