MC504 - Sistemas Operacionais

Processos e Threads
Exclusão mútua

Profa. Islene Calciolari Garcia

Primeiro Semestre de 2017

Sumário

Acesso a recursos compartilhados

- Exemplos de recursos a serem compartilhados: estruturas de dados, arquivos, dispositivos etc.
- Estudo de caso simples com variável inteira:

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

/* As threads da aplicação podem ler ou
   escrever na variável s */

s = thr_id;
```

Como assim `volatile`?

- `volatile` indica ao compilador para sempre deixar os valores atualizados em memória (não fazer otimizações que deixem o valor em registradores)
- o modificador `register` indica que esta variável pode ser armazenada em registradores
- veja o código `register-volatile.c` e os códigos gerados em assembly: `register-volatile-O0.s` e `register-volatile-O2.s`

Acesso à região crítica

- Objetivo: atribuição e impressão sem interferência

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

/* Cada thread tentará executar os seguintes
   comandos sem interferência.   */

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

Execução sem interferência

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

Execução sem interferência

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

Execução com interferência

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`

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 algoritmos 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)

Observações importantes

- Para fins didáticos, nas análises a seguir vamos supor que as threads executam as operações exatamente na ordem indicada pelo código.
- Na prática, otimizações feitas pelo computador ou hardware podem alterar esta ordem.

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

Thread 1

```
while (lock != 0);
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
```

- 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 RX,#1
    XCHG RX, lock
    CMP RX, #0
    JNE entra_RC
    RET

deixa_RC:
    MOV lock, #0
    RET
```

- Instrução *exchange* troca atomicamente 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;
```

Thread 1

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

- Veja o código: `alternancia.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`

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;
```