Departamento de Computação

SISTEMAS OPERACIONAIS 1
21270 A

Departamento de Computação
Prof. Kelen Cristiane Teixeira Vivaldini

Apresentação baseada nos slides
do Profa. Islene Calciolari Garcia
MC504 - Sistemas Operacionais

Departamento de Computação

Sumario

- Condicão de corrida
- Exclusao mútua

- **Condição de corrida**: Quando dois ou mais processos estão lendo ou escrevendo dados compartilhados e o resultado final **depende** de **qual processo executa** e quando (em que ordem) este executa.
- **Exclusão mútua**: impedir que dois ou mais processos acessem um mesmo recurso no mesmo instante, um deve esperar que o outro termine para utilizar.
- **Região crítica**: parte do código onde é feito o acesso ao recurso compartilhado.

Acesso a 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);
```

Acesso a região crítica

volatile

O compilador não pode fazer suposições sobre o valor da variável a qualquer tempo. Ex. O compilador não colocará o valor da variável em um registrador para acesso rápido. Fazer isso incorreria no risco de o valor do registrado não ser o mesmo que o conteúdo da memória variável, que uma interrupção poderia ter alterado após o armazenamento da variável no registrador, sem o conhecimento do programa. Em vez, disso, quando o programa precisar acessar o valor de uma variável, ele especificamente referenciará a posição de memória da variável.

Acesso a região crítica

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 Esperada 1:

Thr 0: 0
Thr 1: 1

Acesso a região crítica

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 Esperada 2:

Thr 1: 1
Thr 0: 0

Acesso a região crítica

volatile int s; /* Vari´avel compartilhada */

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

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

Saída Inesperada:

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 */
}
```

Departamento de Computação

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)

* Starvation: Situação em que um processo nunca consegue executar sua região crítica e acessar o recurso compartilhado.

Departamento de Computação

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

Departamento de Computação

Tentando implementar um lock

- Lock = variável compartilhada com o seguinte significado:
  - lock == 0 ⇒ região critica esta livre
  - lock != 0 ⇒ região critica esta 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 codigo: 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:

le o conteúdo da variável lock;

armazena este conteudo em RX; ´

coloca um valor não nulo em lock.

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

Departamento de Computação

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;
```

Veja o código: interesse2.c

Departamento de Computação

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, entrara 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 é 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;
```

Departamento de Computação

Limitações do algoritmo anterior

- O algoritmo anterior garante exclusão mútua?
- E possível que uma thread ganhe sempre a região critica 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;
```