

MO806/MC914
Tópicos em Sistemas Operacionais
2s2006

Processos e Threads 2

Gerência de recursos para processos e threads

Compartilhados	Independentes
Espaço de endereçamento	Contador de programa
Variáveis globais	Registradores
Arquivos abertos	Pilha
Processos filhos	

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:

```
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

```
int s = 0; /* 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

```
int s = 0; /* 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 disputa

Saída inesperada

```
int s = 0; /* 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`

Tentando implementar um lock

- Considere uma variável compartilhada com o seguinte significado:
 - `lock == 0` $\Rightarrow$ região crítica está livre
 - `lock != 0` $\Rightarrow$ região crítica está ocupada
- Toda thread deve esperar por `lock == 0` para fazer acesso aos recursos compartilhados

Tentando implementar um lock

```
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

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

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 está 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

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

Thread 0

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

Thread 1

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

Algoritmos de Dekker e Desempate

- Como estender estas idéias para N threads?
- 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)

Sugestão para o algoritmo de Dekker

```
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] = 0;
```