Programação Concorrente

```
Procedure sort ( int low, high )
{
    for i = low to high - 1 do
            for  j = i + 1 to high do
                    if a[ j ] < a [ i ]
                    {
                            temp = a [ j ]
                            a[ j ] =  a[ i ]
                            a [ i ] = temp
                    }
}
begin
            for k = 1 to n do
                    read ( a [ k ] )
            sort( 1, n )
            for k = 1 to n do
                    write ( a [ k ] )
end.
```


Ir p/ primeira página

Programação Concorrente

- **Custo do algoritmo: $n^2/2$**

- **Paralelamente: (2 processos)**
  - **$n^2/8$ (ordenar)**
  - **n (merge)**
  - **$n^2/4 + n$ (total)**

- **begin**
  sort (1, n)
  sort (n+1, tam)
  merge (1, n+1, tam)
  end.

N	$n^2/2$	$(n^2/4)+n$	$n^2/8+n$
40	800	440	140
100	5000	2600	1350
1000	500 000	251 000	126 000

Ir p/ primeira página

Programação Concorrente

```
Program sort
{
    (* entra aqui as rotinas anteriores *)

    for k = 1 to tam do
            read ( a[ k ] )
    cobegin
            sort(1, n)
            sort(n+1, tam)
    coend
            merge(1, n+1, tam)
}
```

- cobegin / coend:
  - rotinas podem ser executadas em paralelo
  - o sistema de hardware/software decide

Ir p/ primeira página

Programação Concorrente

- **Concorrente:**
  - **processos com potencial para execução paralela**
- **Programação Concorrente:**
  - **Notações e técnicas para expressar potencial paralelismo**
  - **resolver problemas de sincronização e comunicação**
- **Problema básico:**
  - **que atividades executar concorrentemente**

```
cobegin
        sort( 1, n )
        sort( n+1, tam )
        merge( 1, n+1, tam )
coend
```

Ir p/ primeira página

Programação Concorrente

	P1	P2	P3
Inicial	4, 2, 7, 6, 1	8, 5, 0, 3, 9	-
P1	2, 4, 7, 6, 1	8, 5, 0, 3, 9	-
P2	2, 4, 7, 6, 1	5, 8, 0, 3, 9	-
merge	“	“	2
merge	“	“	2, 4
merge	“	“	2, 4, 5

Merge só será executado corretamente se existir uma forma de sincronização com Sort

Ir p/ primeira página

Exclusão Mútua

- **Abstração para problemas de sincronização**

Sejam A1 e A2 atividades relativas aos processos P1 e P2

- **A1 e A2 são mutualmente exclusivas se não puderem se sobrepor**

ou seja,

se P1 e P2 tentam executar simultaneamente suas atividade A_i

então

apenas um processo pode ter sucesso

Ir p/ primeira página

Exclusão Mútua

- A abstração é expressa como a sequência:

  comandos
  pré-protocolo
  seção crítica
  pós-protocolo

- Definições:
  - Deadlock
  - Lockout / Starvation
  - Atomicidade

Ir p/ primeira página

Exclusão Mútua

Consideramos que o problema de exclusão mútua em alto nível fará uso de recursos fornecidos por níveis mais baixos (hardware)

- **Exemplo**

  n = n + 1

  Load n ← **atômica**
  Add 1
  Store n

Se n = 3 e P1, P2 executam concorrentemente qual o valor final de n?

Ir p/ primeira página

Soluções para Exclusão Mútua

- **Busy waiting**
  - **um processo aguarda sua vez de entrar em sua seção crítica**
  - **testando periodicamente se algum outro processo já executa uma seção crítica**

Genericamente

```
Processo 1
inicio
    while  ( outro processo em seção crítica)  do
                        (* nada *)  ;
    seção crítica
    comandos
fim
```

Ir p/ primeira página

Soluções para Exclusão Mútua

```
procedure p1;
repeat
        while   turn = 2   do   (* nothing *) ;
        critical1;
        turn = 2;
        comandos
forever

procedure p2;
repeat
        while   turn = 1   do   (* nothing *) ;
        critical2;
        turn = 1;
        comandos
forever

turn = 1
cobegin
        p1; p2;
coend
```

Ir p/ primeira página

Soluções para Exclusão Mútua

- Esta solução:
  - resolve exclusão mútua
  - a princípio não permite deadlock
  - nem lockout

- Problema
  - Se P1 tem que ser executado 100 vezes por dia e P2 apenas uma vez como resolver?
  - Se P2 "aborta" antes de passar a vez?

Ir p/ primeira página

Segunda Tentativa

```
procedure p1;
repeat
        while   c2 = 1   do  (* nothing *) ;
        c1 = 1;
        critical1;
        c1 = 0;
        comandos
forever

procedure p2;
repeat
        while   c1 = 1   do  (* nothing *) ;
        c2 = 1;
        critical2;
        c2 = 0;
        comandos
forever

c1  =  c2  =  0;
cobegin
        p1;   p2;
coend
```

Ir p/ primeira página

Problemas

- **Não garante exclusão mútua:**

	C1	C2
inicialmente	0	0
P1 verifica C2	0	0
P2 verifica C1	0	0
P1 seta C1	1	0
P2 seta C2	1	1
P1 na seção critica	1	1
P2 na seção critica	1	1

Como ambos estão na seção crítica, o programa está incorreto!

Ir p/ primeira página

Terceira Tentativa

- **Na solução anterior o erro está na inicialização de c1 e c2 após o teste que valida P1 ou P2 em sua região crítica**

- **Vamos então definir:**

```
procedure p1;
repeat
        c1 = 1;    <-   intenção de entrar na seção
                                crítica
        while   c2 = 1   do  (* nothing *) ;
        critical1;
        c1 = 0;
        comandos
forever
```

Ir p/ primeira página

Terceira Tentativa

```
procedure p2;
repeat
        c2 = 1;
        while   c1 = 1   do  (* nothing *) ;
        critical2;
        c2 = 0;
        comandos
forever


c1  =  c2  =  0;
cobegin
        p1;
        p2;
coend
```

Ir p/ primeira página

Problemas

- **Infelizmente esta solução leva a uma situação de Deadlock**

	C1	C2
inicialmente	0	0
P1 seta c1	1	0
P2 seta c2	1	1
P1 verifica c2	1	0
P2 verifica c1	1	1

O programa garante exclusão mútua mas eventualmente causa DEADLOCK

Ir p/ primeira página

Quarta Tentativa

```
procedure p1;
repeat
        c1 = 1;
        while   c2 = 1   do
        begin
            c1 = 0;     <-    o processo procura dar
                              uma chance ao outro
            (* do nothing *)
            c1 = 1;
        end
        critical1;
        c1 = 0;
        comandos
forever
```

Ir p/ primeira página

Quarta Tentativa

```
procedure p2;
repeat
        c2 = 1;
        while   c1 = 1   do
        begin
                c2 = 0;
                (* do nothing *)
                c2 = 1;
        end
        critical2;
        c2 = 0;
        comandos
forever


c1  =  c2  =  0;
cobegin
        p1;  p2;
coend
```

Ir p/ primeira página

Quarta Tentativa

- Infelizmente a solução proposta, embora garanta exclusão mútua, não está correta

	C1	C2
inicialmente	0	0
P1 seta c1	1	0
P2 seta c2	1	1
P1 verifica c2	1	1
P2 verifica c1	1	1
P1 seta c1	0	1
P2 seta c2	0	0
P1 seta c1	1	0
P2 seta c2	1	1

- Classificação: lockout (em algum momento algum processo pode seguir executando)

- Note que este exemplo exigiria um perfeito sincronismo para acontecer

Ir p/ primeira página

Algoritmo de Dekker

```
procedure p1;
repeat
    c1 = 0
    while   c2 = 0   do
            if turn = 2  then
            begin
                    c1 = 1
                            while turn = 2 do      ;
                    c1 = 0
            end
    critico1
    turn = 2
    c1 = 1
    comandos
forever

begin
    c1  =  c2  =  turn  =  1
    cobegin    p1;    p2;    coend;
end
```

Ir p/ primeira página

Algoritmo de Dekker

- Combinação da 1ª e 4ª tentativas
- lembrando que:
  - 1ª tentativa: deadlock
  - 2ª tentativa: lockout
- Idéia
  - Resolver o problema de lockout dando direi-to (passando a vez) do outro processo in- sistir na entrada da seção crítica

Ir p/ primeira página

Algoritmo de Dekker

- **Funcionamento**
  - **Se P1 seta *c1 = 0* e descobre que P2 setou *c2 = 0* então P1 consulta *Turn***
  - **Se turn = 1 então P1 sabe que é sua vez de insistir na entrada:**
    - **P1 verifica periodicamente c2**
  - **P2 nota que é sua vez de ceder e gentilmente seta c2 = 1:**
    - **o que será percebido por P1**
  - **P2 espera P1 terminar sua seção crítica:**
    - **P1 seta c1 = 1**
    - **Reseta turn = 2**
    - **transferindo, agora, o direito para P2**

Ir p/ primeira página

Conclusão

- **Exclusão mútua:**
  - **é um dos problemas mais simples em P.P.**
  - **difícil de se obter uma solução correta**
  - **necessita de uma solução mais expressiva**
    - **diferente de árbitros de memória**

- **Uso de árbitros de memória / busy waiting**
  - **perda de tempo de CPU**

- **Solução**
  - **uso de primitivas que suspendam a execução de um processo bloqueado**

Ir p/ primeira página

Semáforo

- Simples de implementar
- Elegante na resolução de P.P.
- É um inteiro que assume valores não negativos
- Únicas operações permitidas
  - wait (s) e signal (s)
- wait (s)

  se s > 0
  então s = s - 1
  senão o processo é suspenso
- signal (s)

  se algum processo P está suspenso
  então acorde P
  senão s = s + 1

Ir p/ primeira página

Semáforos

- **Note que:**
  - **wait(s) e signal(s) são operações primitivas como Load e Store**
  - **São mutualmente exclusivas**
    - ☞ **quando atuam no mesmo semáforo**
  - **signal(s) não especifica que processo é acordado quando mais de um processo está suspenso no mesmo semáforo**

Ir p/ primeira página

Exclusão Mútua

```
s : semáforo

procedure p1
repeat
        wait ( s )
        critico 1
        signal ( s )
        comandos
forever

procedure p2
repeat
        wait ( s )
        critico 2
        signal ( s )
        comandos
forever

begin
        s = 1
        cobegin   p1;    p2;    coend
end
```

Ir p/ primeira página

Etapas

- Um processo executa wait (s)
  - se s = 1
    - faz s = 0 e entra na seção crítica
  - caso contrário
    - suspende o processo
- Após a execução da seção crítica
  - o processo sinalizador faz s = 1 e
  - acorda os demais processos (ou 1 dos processos) se estes existirem
- Esta solução é similar as anteriores só que:
  - o teste e alterações de s estão encapsulados em instruções primitivas
  - se p1 notar que s = 1 ele setará s = 0 antes de p2 ter a chance testar o valor de s

Ir p/ primeira página

Exclusão Mútua

- No algoritmo anterior prova-se a ausência de deadlocks e a garantia de exclusão mútua

- Para um problema com n processos temos:

```
const  n =  número de processos
var    s  :  semáforo  (* binário *)

procedure process ( i : integer )
repeat
        wait ( s )
        critico
        signal ( s )
forever

begin
        s = 1
        cobegin   p ( 1 )   ....   P ( n )  coend
end
```

Ir p/ primeira página

Produtor X Consumidor

- problema é causado:
  - pela necessidade do produtor armazenar dados até que o consumidor esteja pronto
  - pelo consumidor que não pode processar dados que não foram gerados pelo produtor
- solução: rendez-vous
- problema:
  - se taxa de produção e consumo varia
  - buffer: armazenar dados
- buffer:
  - área de memória compartilhada

Ir p/ primeira página

Produtor X Consumidor

- considerando um buffer ilimitado temos:

```
repeat
        produz   R
        b [ in ]  =  R
        in++
forever


repeat
        espere   até    que   in > out
        W  =   b [ out ]
        out++
        consume  W
forever
```

Ir p/ primeira página

Produtor X Consumidor

- considerando S = in - out
- S representa o número de registros no buffer
- valores de S:
  - inicialmente 0
  - cresce e decresce arbitrariamente exceto quando S = 0:
    - o consumidor se recusa a decrementar S
    - neste caso espera pelo produtor
- forçando um consumo imediato do registro produzido:
  - 0 = (in - out) = (in + 1) - (out + 1)

S se comporta como um semáforo

Ir p/ primeira página

Produtor X Consumidor

- **Note:**

  repeat
  (1) espere até que in > out
  : : :
  forever

- **podemos transformar facilmente (1) em:**
  - **wait (S)**
  - **assumindo que signal (S) é incluído no produtor**

Ir p/ primeira página

Produtor X Consumidor

```
Var
    n :  semaforo   (* genérico *)

produtor ( )                          consumidor ( )
    repeat                                repeat
       produz                                wait ( n )
       insere no buffer                retira do buffer
       signal ( n )                           consume
    forever                               forever


          n  =  0
          cobegin
                    produtor;    consumidor;
          coend
```

Ir p/ primeira página

Produtor X Consumidor

- assumindo que a inserção e retirada são críticas

```
Var
    n :  semaforo   (* genérico *)
    s :  semaforo   (* binario  *)

produtor (   )                    consumidor (   )
    repeat                            repeat
       produz                            wait ( n )
       wait ( s )                        wait ( s )
       insere no buffer                  retira do buffer
       signal ( s )                      signal ( s )
       signal ( n )                      consume
    forever                           forever


          n   =   0
          s   =   1
          cobegin
                    produtor;    consumidor;
          coend
```

Ir p/ primeira página

Produtor X Consumidor

- o que acontece se trocarmos signal(s) signal(n) por signal(n) signal(s)
- e se trocarmos wait(n) wait(s) por wait(s) wait(n)?
- o exemplo mostra uma fraqueza no uso de semáforos:
  - não é possível entrar ou sair condicional-mente de um wait
  - não é possível examinar seu valor sem executar um wait:
    - neste exemplo entramos em deadlock

Ir p/ primeira página

Produtor X Consumidor

- usando apenas 1 semáforo binário e obviamente uma variável para contar os registros temos:

```
Var      n        : integer
         delay    : semaforo   (* binario *)
         s        : semaforo   (* binario  *)

produtor (   )
    repeat
            produz
            wait ( s )
            insere no buffer
            n   =   n  +  1
            if  n  =  1   then signal  ( delay )
           signal ( s )
    forever
```

Ir p/ primeira página

Produtor X Consumidor

```
consumidor (   )
var  m : integer
     wait ( delay )
     repeat
          wait ( s )
          retira do buffer
          n  =  n  -  1
          m  =  n
          signal ( s )
          consuma
          if  m = 0 then wait ( delay )
     forever

n   =   0        s   =   1     delay  =  0
cobegin
     produtor;
     consumidor;
coend
```

Ir p/ primeira página

Produtor X Consumidor

- note que:
  - wait (delay) não permite que o consumidor execute com o buffer vazio
  - M é usado para testar o valor de N como se estivesse na seção crítica
    - se trocarmos if n = 0 then wait (delay) teríamos:

Ação	n	delay
inicialmente	0	0
produtor	1	1
consumidor	0	0
produtor	1	1
consumidor	0	1 (*)
consumidor	-1	0

Ir p/ primeira página