Prog. Concorrente 438 17/2/9

(www.ime.usp.br/~gold) Alfredo Goldman

monitores: Eduardo Katayama
Hugo Corbucci

Multithreaded, Parallel, and Distributed Programming - Gregory Andrews

Principles of Concurrent and Distributed Programming
M. ~~Ben~~-Ari - 2nd edition

P1 - ~~12/05~~ 14/04
P2 - 16/06
SUB - 30/06 ~~fechada~~ quem faltou / quem não passou

Break { 23/02
18/05 → aula concorrentes
6/04

3EPS, MEP = (EP1 + EP2 + 2*EP3)/4

Pattern for distributed and concurrent programming. D. Lea

Sexta 8:15

Panorama da Computação Concorrente

importância crescente: { multi-core / cloud computing

- máquinas com diversos procesadores
- rede de computadores
- máquinas mais rápidas

? que é ?

Programa concorrente contem dois ou mais procesos que colaboram para cumprir uma tarefa.

Os procesos colaboram através da comunicação
{ variáveis compartilhadas / troca de mensagens

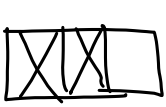
Comunicação ⇒ sincronização

- tipos básicos

exclusão mútua (seções críticas)

↳ sincronização de condição (atrasa um processo até que uma condição seja verdadeira)
→ "pergunta"

Problema clássico: produtor/consumidor



mesma trava: garante que nada mude entre olhar e colocar

Um pouco de arquitetura

máquinas sequenciais

memória principal

cache nível 2

cache nível 1

UCP

diferenças de velocidade

registrador - 1 ciclo (1/10)

± 10 ciclos (1K) (1M)

1 ou 2 ciclos (1K)

disco - 10K ciclos

Máquinas sequenciais

memória principal
cache nível 2
" " 1

UCP

diferenças de velocidade

local	ciclos	quantidade
registradores	1	unidade / dezena
cache 1	1 ou 2	milhares
" 2	± 10	milhões
memória	50 a 100	centenas de milhões
disco	10k	

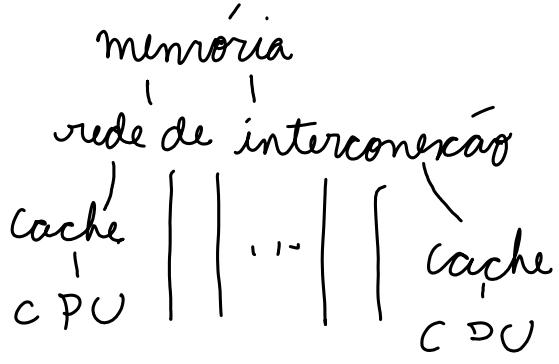
Sistemas multitarefa

REG | SO | REG | Prog 1 | REG | Prog 2 | ...

REG | UCP

Troca de contexto

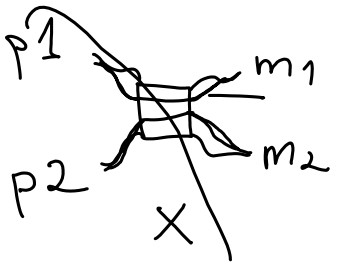
máquinas com memória compartilhada



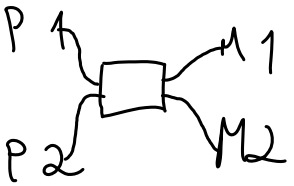
rede = barramento

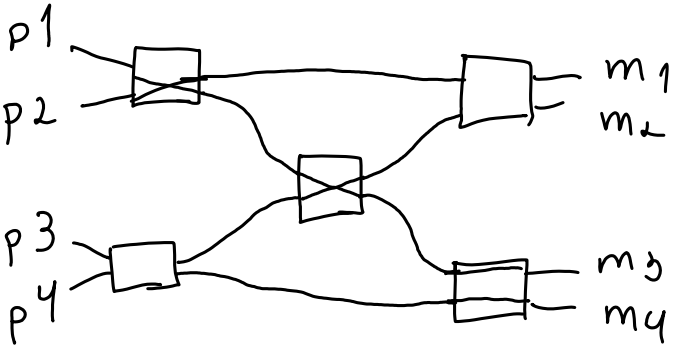
crossbar switch

=



= ou X

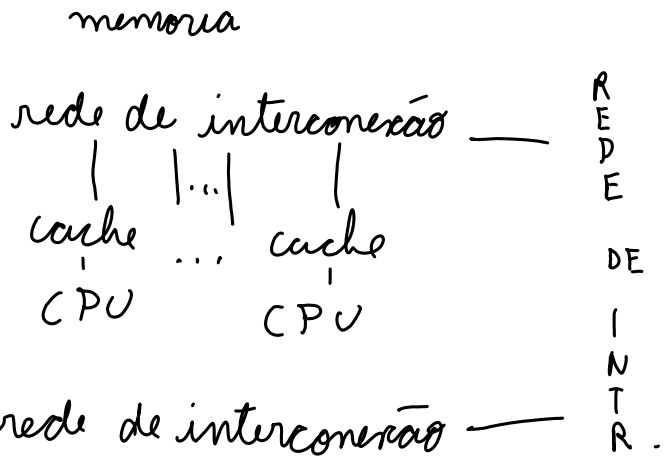




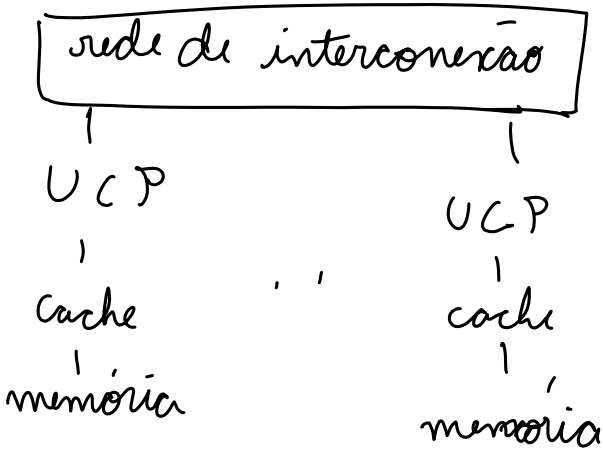
onc. 20/2 L-5

UMA (Uniform Memory Access)

máquinas NUMA



máquinas com memória distribuída

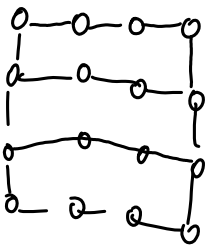


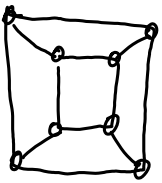
comunicação: troca de mensagens

pode-se usar memória compartilhada virtual

máquina paralela x cluster
- maior velocidade na comunicação
- mais cara

rede de interconexão rápida e dedicada
Exc- grade / hipercubo





aglomerado (cluster)

placas de rede comuns

grade { rede entre aglomerados
Internet

3 tipos principais

multi-tarefa
- SO (sistema de janelas)
- apl. tempo real
- modelo de eventos

sistemas distribuídos
- BDs
- Sistema de arquivos
- Sistemas tolerantes a Falhas (redundância)
- Servidor WEB

computação paralela
- computação ~~paralela~~ científica
- coisas gráficas (Pixar)
- exploração de petróleo

Paradigmas de programação conc. 3/3

concorrente

1) Paralelismo iterativo
2) Paralelismo recursivo
3) Sistemas produtor/consumidor
4) Clientes e servidores
5) Iteração entre pares

Paralelismo iterativo

Quando um programa contem diversos processos, cada um com um ou mais laços. Os processos cooperam para resolver um único problema.

cooperação → comunicação → sincroniza

Ex.: multiplicação de matrizes

```
double a[n,n], b[n,n], c[n,n];
for [i=0 to n-1] {
    for [j=0 to n-1] {
    |   c[i,j] = 0.0;
    |   for [k=0 to n-1]
    |       c[i,j] = c[i,j] + a[i,k] *
                                b[k,j];
    }
}
```

trivialmente paralelizado

Independência

Definição: duas operações são independentes se o conjunto de variáveis alterados (e lidas) são disjuntos.

- trocar os 2 primeiros FORs por CO. = sincronização implícita. Começa a execução em paralelo e aguardo o seu término E se n é muito grande?

"custo de criação de thread maior que - de multiplicação. Logo não vale a pena fazer uma multiplicação em cada thread $O(n \cdot n^2)$ "

" problema de o cache carregar bloco que será invalidado por outro " Solução: trocar os COs de i com j (linhas) e colocar FOR no i.

~~em múltiplos chamados~~

primitiva procer ≅ co executado em background

pode ser interessante criar apenas P procesor

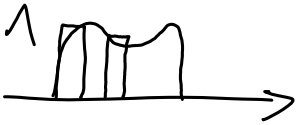
```
procer worker [w = 1 to P]
| int f = (w - 1) * n/P;
| int l = f + n/P - 1;
  for [i = f to l] {
|     for [j = 0 to n-1] {
          c[i,j] = 0 0;
|         □  > último for do anterior
|     }
}
```

-) Paralelismo recursivo

Um programa recursivo pode ser implementado usando concorrência quando

tem múltiplas chamadas recursivas independentes.

Ex: Cálculo de integral



versão iterativa

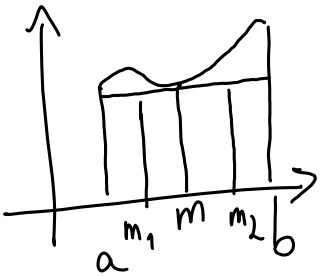
```
double fleft = f(a), fright, area = 0.0;
double largura = (b-a).MAX;
for [x = (a + largura) to b by largura] {
    fright = f(x);
    area = area + (fleft + fright) * largura/2;
    fleft = fright;
}
```

método divisão e conquista

- ache o ponto médio m
- ache as áreas a-m, m-b e a-b

re estiver proximo Ok

c.c. continua




```
double quad (double left, right, fleft,
             right, larea) {
  double mid = (left + right)/2.0;
  double fmid = f(mid)

  double larea = (fleft + fmid)* (mid-left)
                                  /2.0;
  double rarea = ( fmid + fright) *
                 (right - mid)/2.0;
  if ( |larea + rarea - larea| > E) {
                        brarea
    larea = quad (left, mid, fleft, fmid,
```

```
        larea);
        rarea = quad(mid, right, fmid, fright,
                                    rarea);
    }
    return (larea + rarea);
}
```

CO no bloco de cálculo de quadr.

problema: possível excesso de concorrência

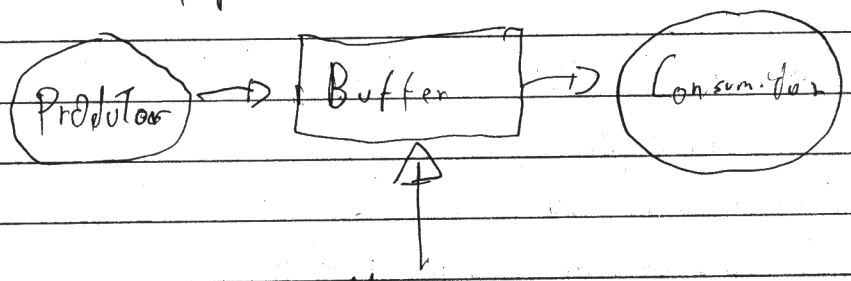
redução: parâmetro a mais = profundidade.

problema: quando um lado se aprofunda muito mais que outro

outros exemplos: ordenação, n-ranhas

3) Produtor / Consumidor

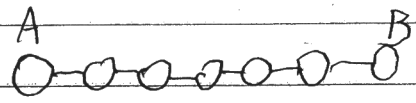
Um produz, outro consome, podem estar organizados em uma pipeline.



Níveis críticos
Cheio / Vazio

Exemplos: Unix Pipes
cat lixo.txt | sort | uniq

Transmissão de Mensagens em Caminhos



Tamanho L
$\alpha + L\gamma$.

M

α - Start up
γ - Inverso da banda passante

Dividir a mensagem em p pedaços

$$\alpha + \frac{L}{p}\gamma + \alpha + \frac{L}{p}\gamma$$

4) Cliente / Servidor

Padrão dominante em sistemas distribuídos.
Cliente: Solicita serviços, espera pela resposta
Servidor: Aguarda e processa pedidos (1 por vez ou multi-thread) se em máquinas diferentes usa RPC.

5) Interação entre pares

Ocorre em programas distribuídos quando existem vários processos que executam praticamente o mesmo código, e trocam mensagens para realizar uma tarefa

Exemplos: multiplicação de matrizes distribuída

⇒ troca de mensagens

Mestre-escravo

Calcular $a \times b$ onde a e b são matrizes $n \times n$ usando n processos um por máquina

```
process worker [i = 0 to n-1] {
    double a[n]; # linha i da matriz a
    double b[n,n]; # matriz b
    double c[n]; # linha i da matriz c
    receive (valores de a e b)
    for [j = 0 to n-1] {
        c[j] = 0.0;
        for [k = 0 to n-1]
            c[j] = c[j] + a[k] * b[j,k];
    }
    send (vetor c);
}
```

```
process coordinator {
    double a[n,n];
    double b[n,n];
    double c[n,n];
    inicializa a e b
    for [i=0 to n-1] {
        send(linha i de a para worker[i]);
        send(matriz b para worker[i]);
    }
    for [i=0 to n-1]
        receive(linha i de c do worker[i]);
    imprime resultados
}
```

send/receive - primitivas para troca de mensagens Bloqueantes

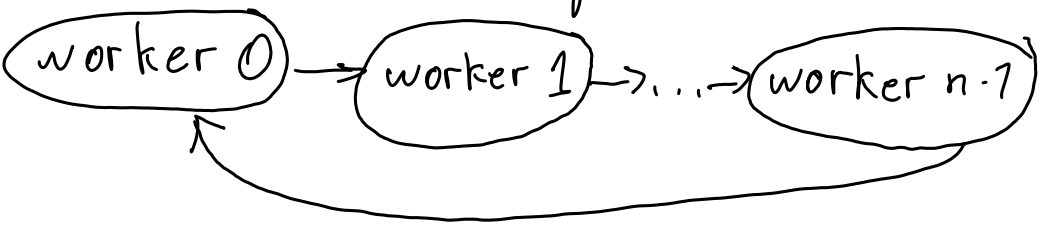
coordinator: nor nindr, fazer o primeiro com CO e colocar fora o segundo, trocando o por broadcast

broadcast - difusão

gosiping - troca completa

Outro exemplo: pipeline circular

Suponha que inicialmente cada processo possua apenas a linha i de a e a coluna i de b. Para calcular a solução, as colunas de b têm que circular



```
process worker [i=0 to n-1]{
    double a[n] b[n], c[n];
    double sum = 0.0,
    int vertcol = i;
    receive (linha i de a e coluna i de b)
    # calcula c[i,i]
    for [k=0 to n-1]
        sum = sum + a[k]*b[k];
    c[vertcol] = sum;
    for [j=1 to n-1] {
        send (coluna b para o próximo worker)
        receive (coluna b do worker anterior)
        sum = 0.0;
        for [k=0 to n-1]
            sum += a[k]*b[k];
        if (vertcol == 0)
            vertcol = n-1;
```

```
    else
        nertcol = nertcol - 1;
    c[nertcol] = num;
  }
  send(c para o coordenador);
}
```

Programação com variáveis compartilhadas 10/03
(contexto de memória compartilhada → multicore por exemplo).
Capítulo 2 { Andrews
Ben-Ari.

Definição: Um programa concorrente consiste de um número finito de processos. Cada processo é escrito usando um conjunto finito de instruções atômicas (indivisíveis).

Definição: O estado de um programa é o conteúdo de suas variáveis em um dado momento.

variáveis { explícitas
implícitas { registradores
ponteiro de controle (onde tá a base e o topo da pilha, PC = program counter)

No contexto concorrente, cada processo tem seu PC.

ponteiro de controle { - único para cada processo.
- indica a próxima instrução a ser executada.

um processo executa uma sequência de instruções.

instruções: sequência de uma ou mais ações atômicas { indivisível
programa e/ou altera o estado do programa.

A execução de um programa concorrente consiste de sequências de instruções atômicas intercaladas (= não tem execução simultânea)

História uma sequência de execução de ações atômicas.

Exemplo: 3 processos P: p1, p2, p3, p4 cada processo com 4
Q: q1, q2, q3, q4 ações atômicas.
R: r1, r2, r3, r4.

Aqui tem muitas histórias possíveis

Algumas delas são inválidas por exemplo: não pode fazer q3 antes de p1 (porque p1 altera algo que q3 olha) Como faz isso?
(colocar várias instruções antes que q1 não é solução → não funciona sempre).
Para isso, vamos usar uma primitiva, uma barreira.

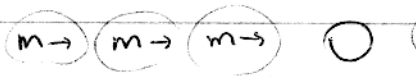
Quais as possíveis execuções de P: p1, p2 ? { p1 p2 q1 q2 ; q1 q2 p1 p2 ; p1 q1 p2 q2
Q: q1, q2.

O compilador tem o direito de olhar as instruções p1, p2 e observar que elas são independentes e executar p2 antes de p1
Então pode acontecer p2 q1 q2 p1
O processador também pode mudar a ordem do código compilado

Papel da sincronização ⇒ restringir o conjunto de histórias possíveis a um conjunto desejável. Garantindo que q3 não seja executada antes de p1
2 formas: I) exclusão mútua "amarra" ações atômicas
Ex: P: <p1, p2>, p3, p4
Significa que p1 e p2 são atômicas. p1 executa p1, logo tem que executar p2. Se p1 coloca 1000 numa variável e p2 coloca zero, nenhum outro processo vê esse 1000 [criar regiões que "pareçam" atômica].

II) Condição de sincronização espera alguma coisa acontecer para continuar [atrasar uma ação até que seja válida uma condição].

Exemplo Frog Puzzle.



dois movimentos possíveis: 1) pular ao espaço vazio ao lado.
2) pular um sapo e cair em um espaço vazio.

• Dá pra chegar no estado : m f f ○ m m m ?

Imagina cada sapo como um processo e vê quais os estados possíveis:

inicial: m m ○ f f

1) ○ m m f f travou: estado final
2) m ○ m f f
3) m m f ○ f
4) m m f f ○ travou: estado final.
5) m f m ○ f ←...
6) m ○ f m f ←...

Análise final de estados de concorrência. Dá pra ver a chance de chegar no estado final desejável. (nesse caso ⇒ 1/6 aprox)

Exemplo: n = 0;
co {
n = n+1; ①
// n = n+1; ②
}

Qual o valor final de n?

Se exclusão mútua : n = 2

Pode dar 1 se quebrar a instrução.

Mas também pode dar qualquer coisa: se interrompe a soma no meio. O outro vai acessar o n onde tem uma soma feita parcialmente

safe

Outro exemplo:

```
n = 0;
co {
    temp1 = n;
    n = temp1 + 1;
// temp2 = n
    n = temp2 + 1;
}
```

No final: pode ser 2; pode ser 1.

Assume-se que a atribuição é atômica. Então não pode ter qualquer coisa → só vai atribuir ali se a soma estiver completa.

Propriedades de um programa

atributo verdadeiro para todas execuções parciais.

dois tipos:

- safety - o programa nunca entra em um estado ruim (estoura a pilha por exemplo)
  ex: em SO o mouse sempre é tratado
- liveness - em algum momento o programa entra em um estado desejável (prog acaba)
  ex: em SO ao clicar temos uma resposta

exemplo de safety: corretude parcial: se o programa termina ele produz a resposta certa

liveness: toda história possível termina

Corretude TOTAL: os dois juntos → O programa sempre termina e em um estado desejável

Em um contexto de Concorrência

- safety: exclusão mútua
- liveness: entrada na seção crítica

Formas de verificar as propriedades:

- Teste/Depuração ⇒ verificar um nº limitado
- Análise Exaustiva ⇒ existem geral/e MUITAS
- Análise Abstrata

histórias

13/03

Exemplo: Paralelização da busca de padrões em arquivo

grep "padrão" [arquivo].

Sequencial:

```
string line;
lê uma linha da entrada em line.
while (! EOF) {
    procure o padrão em line
    if ( padrão ∈ line)
        write line
    leia a próxima linha
}.
```

Procurar partes independentes.

Definição: Seja o conjunto de leitura de um programa as variáveis lidas, mas não alteradas. Seja o conjunto de escrita as variáveis que são gravadas (e possivelmente lidas). Duas partes de um programa são independentes se a interseção do conjunto de escrita de cada uma delas é disjunto dos conjuntos de leitura e escrita da outra.

As vezes é possível que dois processos possam ser executados em paralelo, mesmo com gravação nos mesmos elementos, mas isto só é possível apenas quando esta ordem não é importante.

Possível concorrência:

Formas de verificar as propriedades:

- Teste/Depuração ⇒ verificar um nº limitado
- Análise Exaustiva ⇒ existem geral/e MUITAS
- Análise Abstrata

histórias

13/03

Exemplo: Paralelização da busca de padrões em arquivo

grep "padrão" [arquivo].

Sequencial:

```
string line;
lê uma linha da entrada em line.
while (! EOF) {
    procure o padrão em line
    if ( padrão ∈ line)
        write line
    leia a próxima linha
}.
```

Procurar partes independentes.

Definição: Seja o conjunto de leitura de um programa as variáveis lidas, mas não alteradas. Seja o conjunto de escrita as variáveis que são gravadas (e possivelmente lidas). Duas partes de um programa são independentes se a interseção do conjunto de escrita de cada uma delas é disjunto dos conjuntos de leitura e escrita da outra.

As vezes é possível que dois processos possam ser executados em paralelo, mesmo com gravação nos mesmos elementos, mas isto só é possível apenas quando esta ordem não é importante.

Possível concorrência:

busca e leitura simultânea.

```
while (!EOF) {
    co {
        procura o padrão em line
        if (padrão ∈ line)
            write line
        // leia a próxima linha
    }
}
```

busca e leitura são independentes, mas busca e leitura na mesma variável não são independentes → string line

Então isso tá errado

```
string line1, line2;
leia uma linha em line1;
while (!EOF) {
    co {
        procure o padrão em line1
        if (padrão ∈ line1)
            write line1;
        // leia a próxima linha em line2;
    }
    line1 = line2;
}
```

← sincronização implícita: só pode jogar line2 em line1 quando acabar a busca e a procura. (→ semântica do co)

Problema: procurar padrão e ler linha é rápido. Aqui temos o ~~tempo~~ custo de criar processo. Custo de criar e destruir processo o tempo todo, melhor paralelizar de outro jeito.

Como melhorar? Tirar o co do laço.

```
string buffer;
boolean done = false;
co {        # processo 1.
    string line1;
    while (true) {
        wait buffer cheio (ou done == true;)   condição de sincronização - explícita.
        if (done)
            break;
        line1 = buffer;
        signal buffer vazio   manda um aviso pro wait.
        procura padrão na line1
(*)     if (padrão ∈ line1)                    O signal não se perde
            write line1
    }
// # processo 2.
    string line2
    while (true) {
(*)     leia próxima linha em line2
        if (eof) {
            done = true; signal buffer cheio;
            break;
        }
        wait buffer vazio;
        buffer = line2;
        signal buffer cheio;
    }
}
```

Problemas ① depende que tá o programa, ele trava, pq → deadlock no começo os dois processos vão parar no wait.

Eu tenho que liberar um wait na 1ª iteração → libera o wait de buffer vazio

→ solução: colocar um signal buffer vazio antes de qualquer while

(2)

Pode perder a última linha se achar EOF.

Solução: fica ou done == true do wait do 1º processo.
e coloca wait buffer vazio antes do if (EOF) no 2º processo

Encontrar o máximo de um vetor.

Encontrar o máximo em a[n] com elementos positivos e inteiros

Sequencial:
```
int m = 0;
fa [i = 0 to n-1] {
    if (a[i] > m)
        m = a[i];
}.
```

1ª) troca fa por co.
2ª) cria regiões críticas.
3ª) cria regiões críticas de m = a[i]

4ª solução:
```
int m = 0;                                   double check.
co [i = 0 to n-1] {
    if (a[i] > m)   ← ato mais if's melhor.
        < if (a[i] > m)
            m = a[i]; >
}.
```

17/03

Ações atômicas e o comando await.

duas formas de obtermos "ordem" corretas.

- exclusão mútua: combinar ações em blocos atômicos
- condição de sincronização: atrasar a execução até que uma condição seja válida.

Ações atômicas de grão fino são implementadas pelo hardware.

Exemplo: int y = 0, z = 0;

```
co {
    <x = y + z;>
//  <y = 1>; <z = 2>
}
```

quais os valores possíveis de x?
0, 1, 3, 2.

Ações atômicas e o comando await

duas formas de obtermos "ordens" corretas

- exclusão mutua (combinar ações em blocos atômicos
- condição de sincronização - atrasar a execução até que uma condição seja válida.

Ações atômicas de grão fino são implementadas pelo hardware

Exemplo:

```
int y=0, z=0;
co {
    <x = y + 2>;
    //
    <y = 1>, <z = 2>;
}
```

possíveis valor de x:

0, 1, 3, 2 → otimização

Referência crítica: referência para uma variável modificada por outro processo concorrente.

De entre dois processos concorrentes não há referências críticas a execução parece atómica

- nenhum dos valores dos quais um depende muda
- o outro processo não vê valores intermediários

Ex:

```
int x=0, y=0;
co {
    x = x + 1;
    y = y + 1;
}
```

Mas, em geral isto não acontece.

Logo usa-se um requisito mais geral:

- Propriedade no máximo uma vez

uma atribuição $x = e$ satisfaz a propriedade se:

1- e contém no máximo 1 referência crítica, e x não é lido por outro processo, ou

2- e não contém referências críticas, nesse caso x pode ser lido.

Idéia:
Ter execuções que pareçam atômicas com apenas uma referência não é possível saber quando ocorre a atualização

Ex.:

```
int x=0, y=0;
co {
    x = y + 1;
    // ; referência crítica
    y = y + 1;
}
```

vale no máximo 1 vez (0 ou 1) e será consistente no contexto.

```
int x = 0, y = 0
co {
    x = y + 1;        não satisfaz
    //
    y = x + 1;
}
```

```
int y = 1, x, z;
co {
    y = y + 1;
    //
    x = y;
    //
    z = y;
    //
    if (x != z) print ("Bingo")
}
```

não vale

Primitiva para sincronização: await

conceito com o objetivo de criar uma sequência de comandos atômicos

< await (B) S; >

B: condição de atraso

S: comando

Exemplos:

< await (s > 0) s = s - 1; >

x = 0, y = 0;

< x = x + 1; y = y + 1; > → excl. mútua

< await (count > 2); >

a implementação desta primitiva é difícil.

importante: se a condição B respeita a propriedade no máximo uma vez, < await (B); > é equivalente a while (!B).

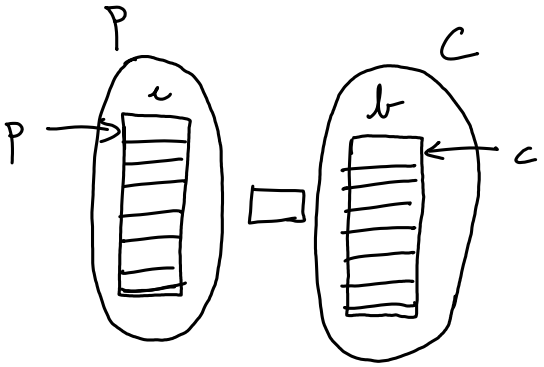
Exemplo:

Produtor / Consumidor com comunicação através de uma variável buffer

Produtor contém um vetor a [n]

Consumidor deve copiar a[n] em b[n]

variáveis p e c que controlam os itens produzidos e consumidos



condição de sincronização

$$0 <= p <= c + 1$$

```
int buf, p = 0, c = 0;
proces Producer {
    int a[n];
    while (p < n) {
        < await (p == c); > while (p > c);
        buf = a[p];
        p = p + 1;
    }
}
```

```
process Consumer {
    int b[n];
    while(c<n){
     < await (p>c); >   while(p==c);
     b[c]=buf;
     c=c+1;
    }
}
```

Propriedades de Safety e Liveness

Safety: não acontece nada errado durante

Liveness: alguma hora algo bom acontece

Programa sequencial

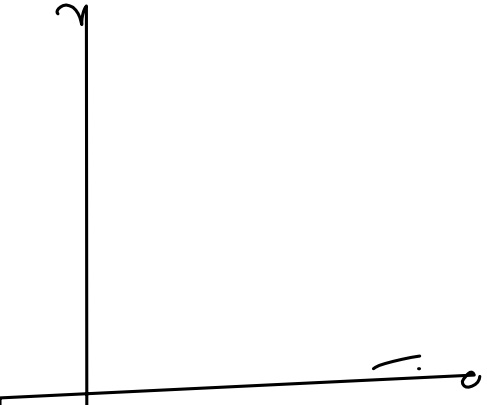
- Safety: estado final correto
- Liveness: chega ao estado final

Programa concorrente

Safety {
- exclusão mútua - não há execução simultânea
- ausência de deadlock - não vai existir a situação "todos esperando"

Livenes:

- Um processo entrará em algum momento



Justiça (Fairness) e políticas de escalonamento

Garantia que todos os processos tem chance de prosseguir

Justiça incondicional - toda ação atômica possível de execução é executada em algum momento (ninguém espera para sempre)

Ex: round-robin

Justiça fraca

- é incondicionalmente justa
- cada ação condicional atômica é executada em algum momento se a condição física fica e permanece verdadeira até que seja vista

Ex.: round-robin, time-slicing

Justiça forte

- é incond. ~~forte~~ justa
- toda ação condicional atômica "executável" é executada em algum momento, assumindo que esta condição é frequentemente verdadeira.

```
boolean continue = true, try = false;
co {
    while (continue.{
        try = true;
        try = false;
    }
    //
    < await (try) continue = false; >
}
```

dá para implementar justiça forte?

não é praticável pois o S.O. teria que entender o programa

Andrews - ex 2.17

Considere:

```
co {
    < await (x >= 3) x = x - 3; >
    //
    < await (x >= 2) x = x - 2; >
    //
    < await (x == 1) x = x + 5; >
}
```

Para que valores de x o programa termina? Com que justiça? Quais os valores finais de x?

incondicional: tenta uma vez somente

fraca: tenta novamente depois que ficar verdadeiro

$x = 1 \Rightarrow x = 1$

6 6

5 } pode acabar, conforme a ordem
4 }

2.33 Considere

```
int x = 10, c = true,
co {
   <await x == 0>; c = false;
   //
   while (c) <x = x-1;>
}
```

a) Termina com justiça fraca?

b) Termina com j. forte!

, e se colocarmos outra barra

```
while (c) { if (x == 0) <x = 10;> }
```

a) não necessariamente

b) depende da linguagem (só se "der a volta" nos inteiros

c) No caso B, termina.

Locks e Barreiras

cap 3 Andrews
cap 3 Ben-Ari

Problema da seção crítica. Neste problema n processos executam repetidamente uma seção crítica e após, uma seção não crítica.

A seção crítica é precedida de um protocolo de entrada e seguida por um protocolo de saída

```
process CS[i = 1 to n] {
    while (true) {
        protocolo de entrada
        seção crítica
        protocolo de saída
        seção normal (não-crítica)
    }
}
```

Suposição: um processo que entra na seção crítica sai dela.

Os protocolos de entrada e saída devem obedecer as propriedades.

Exclusão mútua

SAFETY

Ausência de deadlock - se dois ou mais processos tentam entrar nas suas seções críticas ao menor 1 vai conseguir

Ausência de atraso desnecessário - se um processo está tentando entrar em sua seção crítica, e os outros processos estão executando seções não críticas, ou terminaram, o processo não é impedido de entrar

liveness (Entrada garantida - um processo que está aguardando a entrada na sua seção crítica irá entrar em algum momento.

amor começar com dois processos

1ª tentativa

```
int turn = 1
procss CS1 {
    while (true) {
        <await (turn == 1);>
        seção crítica
        turn = 2;
        seção não crítica
    }
}
procss CS2 {
    :
    <await (turn == 2);>
    seção crítica
    turn = 1;
    :
```

não ocorre ausência de atraso ~
desnecessário (safety), pois um proces-
so pode ser mais longo que outro, au-
mentando a espera do primeiro que não
necessariamente deve rodar em alter-
nância com o outro

Obs.: se tirar a atomicidade do await,
não há exclusão mútua

~ 3ª tentativa

~~boolean in1 = false, in2 = false;~~
~~proc CS1{~~

generalizando:

```
boolean in = false;
process CS i {
    while (true) {
        <await (!in) in = true;>
        seção crítica
        in = false
        seção não crítica
    }
}
```

Obs.: entrada garantida - sim com justiça forte.

"banheiro"

Usar await

1) primitivas de hardware
2) Algoritmos mais sofisticados

instruções do tipo test and-set

fáceis de implementar de forma atômica

```
boolean TS(boolean lock) {
    < boolean inicial = lock;
      lock = true;
      return inicial; >
}
```

```
< await (!in) in = true; >
```

↓

```
while (TS(in))
    skip;
```

← busy waiting

entrada garantida: skip aleatório

otimização para ambientes multi-
processador

```
while (in) skip;
while (TS(in)) {
    while (in) skip;
}
```

para evitar invalidação de cache

Como implementar $\langle S; \rangle$

protocolo de entrada
S;
protocolo de saída

como impl.

<await(B)S;>?

while (!B) - ne vale no máximo 1 vez

proto entrada
while (!B) { protocolo de saída ; skip;
protocolo de entrada } S;
proto saída

outras primitivas de hardware

fetch-and-add

exchange (a, b)

compare-and-swap

```
fetch-and-add (int i)
<
    int val = i;
    i++;
    return val;
>
```

protocolo de entrada while (in != 0);

(FA(in) != 0)

while (in != 0)

skip;

protocolo de saída in = 0;

Otimização para ambientes multiprocessados:

```
while (in) skip;   // este não invalida o cache.
while (TS(in)) {
    while (in) skip;
}
```

* Como implementar $\langle S; \rangle$?

protocolo de entrada

$S;$ ← e S pode ser tão grande quanto eu queira.

protocolo de saída.

* Como implementar $\langle await(B)\ S; \rangle$?

```
while (!B)   // x vale no máximo 1 vez.
    protocolo de entrada
    while (!B) { protocolo de saída; skip; protocolo de entrada; }
    S;
    protocolo de saída.
```

Outras primitivas de hardware:

- fetch-and-add
- exchange(a,b)
- compare-and-swap.

```
fetch-and-add (int i)
< int val = i;
  i++;
  return val; >
```

while (in != 0);

protocolo de entrada: while (FA (in) != 0)
while (in!=0)
skip;
[in=1] dentro da seção crítica, pra evitar que e algumas vezes o int estore.

protocolo de saída: in = 0;

27/03

CS enter
while (!B) { CS exit ; Delay; CS enter }
Seção-crítica
CS exit.

equivalente a <await (B) < seção-crítica >

Soluções justas para o problema da seção-crítica.

- spin lock: testa a condição, espera, testa de novo, espera mais um pouco, até que eu mesmo altero a variável

tem { exclusão mútua
livre de deadlock
sem atraso desnecessário.

mas: entrada garantida? não, só com justiça forte

- Tie-breaker (algoritmo de Peterson)

idéia: os processos devem se revezar quando os dois querem entrar na seção crítica.
Usa uma variável adicional para marcar o último a entrar.

Voltando ao algoritmo mais simples:

	①	②
p. entrada:	while(in2) skip	while(in1) skip;
	in1 = true;	in2 = true.
p. saída:	in1 = false	in2 = false.

Como o protocolo de entrada não é atômico, não há exclusão mútua.

1ª idéia: trocar ~~as~~ ordem → pôr o pé na porta

	①	②
p entrada:	in1 = true;	in2 = true;
	while(in2) skip;	while(in1) skip.

Assim, não existe risco nenhum de entrar 2 processos ao mesmo tempo na seção crítica. Mas pode dar deadlock → possível solução: se os dois estiverem com true, escolhe um pra entrar

Usar variável auxiliar para o caso onde in1 e in2 são verdadeiros

last: variável que indica o último que quer entrar.

bodean in1 = false ; in2 = false int last = 1;

```
process CS1 {                          process CS2 {
  while (true) {                         while (true) {
    in1 = true;                            in2 = true;
    last = 1;                              last = 2;
    < await (!in2 or last==2); >           < await (!in1 or last==1); >
    Seção crítica.                         Seção crítica.
    in1 = false;                           in2 = false;
    seção não-crítica                      seção não-crítica
  }                                      }
}                                      }
```

se trocar, pode dar errado a exclusão mútua.

não resolve deadlock

while(in1 and last==2) skip

pode!

é o tie-breaker

A generalização:

```
int in[1:n] = ([n] 0), last [1:n] = ([n] 0);

process CS[i=1 to n] {
    while (true) {
        for [j=1 to n] {                    // protocolo de entrada
            last[j] = i;        in[i] = j;
            for [k=1 to n such that i!=k] {
                while (in[k] >= in[i] and last[j] == i) skip;
            }
        }
        Seção crítica
        in[i] = 0;              // protocolo de saída
        seção não-crítica;
    }
}
```

- Algoritmo de Dekker.

variável compartilhada turn. que começa com 1.

1ª tentativa.

	①	②
p. entrada:	while(turn != 1) skip.	while(turn != 2) skip.
p. saída:	turn = 2;	turn = 1;

tem atraso desnecessário, lembra?

2ª tentativa.

~~inicio~~ want1 = false , want2 = false;

p. entrada:	while(want 2) skip.	while (want 1) skp.
	want 1 = true;	want 2 = true.
p. saída.	want 1 = false;	want 2 = false.

Falha exclusão mútua, se os 2 testarem ao mesmo tempo.

3ª tentativa.

p. entrada:	①	②
	want 1 = true;	want 2 = true
	while (want 2) skip;	while (want 1) skp;

Dá deadlock.

Ideia: desistir se houver conflito.

p. entrada:

①	②
want 1 = true;	want 2 = true;
while(want 2) {	while (want 1) {
want 1 = false	want 2 = false;
skip;	skip;
want 1 = true;	want 2 = true;
}	}

Falha no atraso desnecessário.

Algoritmo de Dekker

+ variável inteira turn = 1;

① (p. entrada)

```
want1 = true;
while (want2) {
    if (turn == 2) {
        want1 = false;
        while (turn != 1) skip;
        want1 = true;
    }
}
```

② (simétrica)

p. saída: turn = 2; want1 = false.

Funciona!

Algoritmo Manna - Pnueli

want1 = 0, want2 = 0,

p. entrada:

①

```
< if (want2 == -1)
    want1 = -1;
  else want1 = 1; >
while (want1 == want2)
    skip;
```

②

```
< if (want1 == -1)
    want2 = 1;
  else want2 = -1; >
while (want1 == -want2)
    skip;
```

p. saída: want1 = 0; want2 = 0;

Se tirar a atomicidade do if, não funciona.

Ticket Algorithm

//—

Para

EP até amanhã

—//—

ideia: senhar por ordem de chegada

variáveis compartilhadas:

number e next - começam com 1

vetor turn[n] começa com zero

=

0 se não quiser entrar na região crítica

ou ticket caso contrário.

Para entrar na secão crítica o procno CS i faz primeiro { turn[i] = number;
number++;

O procno CS i então espera até que next seja igual ao seu número para en-

trar na seção crítica. Ao sair incre-
menta next.

```
int number = 1, next = 1, turn[1 & n] = ([n]0)
procesr <S [i:= 1 to n] {
    while (true) {
*(      <turn[i] = number; number++;>
        <await (turn[i] == next);>
            seção crítica;
        <next++;>
        seção não crítica.
    }
}
```

* se unir, todos esperam antes e perdem a justiça. Quem chega depois não pega a senha na hora.

intrução Fetch-and-Add

```
FA (var, iner):
< int tmp = var; var = var + iner;
  return (tmp); >

< aux = number++; >
  turn[i] = aux;
```

estouro de inteiro : dificuldade em usar módulo

versão final:

```
procesS[i = 1 to n] {
    while (true) {
        turn[i] = FA (number, 1);
        while (turn != next) skip;
        seção crítica
        next = next + 1;
        seção não crítica
    }
}
```

sem fetch-and-add

Protocolo de Entrada

turn[i] = number; Tie-break

number++;

Protocolo de saída

" região crítica pequena não altera muito a ordem, pois a fila é curta."

" Quando ocorre falta qualquer um pode ser o seguinte"

The Bakery Algorithm

Objetivo: algoritmo sem instruções especiais

Ideia: os clientes verificam entre si quem é o próximo

```
int turn[1:n] = ([n]0);
proces CS[i = 1 to n]{
    while (true) {
        < turn[i] = max (turn[1:n])+1; >
        for[j = 1 to n tal que j ≠ i]
            <await (turn[j] == 0 or turn[i] < turn[j]);>

            região crítica

            turn[i] = 0

            região não crítica
```

problemas:

< turn[i] = max (turn[1:n]) + 1; >

não existe instrução atômica para isto

< await (turn[j] == 0 or turn[i] < turn[j]); >

duas referências a turn[j]

Abordagem: resolver o problema para dois processos.

Protocolo de entrada CS1

turn 1 = turn 2 + 1;

while (turn 2 != 0 and turn 1 > turn 2)

skip

Protocolo de entrada CS2

turn 2 = turn 1 + 1;

while (turn 1 != 0 and turn 2 >= turn 1)

skip;

não vale exclusão mútua

Para resolver os problemas ↖ pé na porta:

logo depois dos protocolos de entrada, colocar

turn = 1

Para generalizar, a condição teria que ser simétrica...

Artifício:

$(a,b) > (c,d)$ == true se $a > c$ ou se $a == c$ e $b > d$

== false caso contrário

generalização

```
int turn[1:n] - ([n] 0);
proc <S[i = 1 to n]{
    while (true) {
        turn[i] = 1;
        turn[i] = max(turn[1:n]) + 1;
        for[j = 1 to n tal que j ≠ i]
        : while (turn[j] != 0 and
        :           (turn[i], i) > (turn[j], j))
        :       skip;
        : ;
        : região crítica
```

Comp. ~~Matriz~~[.]=0; 1/4/09 108

Página 13 das notas de aula. seção não crítica;

} // while

} // process

Comp. Musical 3/4/09

Procuro não pensar

- portas

- Ao entrar na primeira porta, marca-a
- Antes de entrar na segunda, marca-a e verifica a anterior
- Antes de entrar na primeira, olha pela fechadura se a segunda está marcada.

```
integer gate1, gate2;
procesr CS1 {
    while (true) {
        seção não crítica;
        p1: gate1 = p;
        if (gate2 != 0) goto p1;
        gate2 = p;
        if (gate2 != p)
            if (gate1 != p)
                goto p1;
        seção crítica;
        gate2 = 0;
    }
}
```

```
proces CS2{
    while(true){
        seção não crítica;
q1:     gate1 = q;
        if (gate2 != 0) goto q1
        gate2 = q;
        if(gate2 != q)
            if (gate1 != q) goto q1
        seção crítica,
        gate2 = 0;
    }
}
```

não tem exclusão mútua.

versão final.

```
integer gate1 = 0, gate2 = 0;
boolean want1 = 0, want2 = 0;
CS1: laço {
        não crítica
        seção
p1:     gate1 = 1;
        want1 = true;
        if (gate2 != 0) {
            want1 = false;
            goto p1:
        }
        gate2 = 1;
        if (gate1 != 1) {
            want1 = false;
            await want2 == false;
            if (gate2 != 1) goto p1
            else want1 = true;
        }
        seção crítica;
        gate2 = 0;
```

```
    want1 = false;
}
```

CS2:

```
laço {
    seção não-crítica
q1: gate1 = 2;
    want2 = true;
    if (gate2 != 0){
        want2 = false; goto q1:
    }
    gate2 = 2;
    if (gate1 != 2){
        want2 = false;
        await (want1 == false)
        if (gate2 != 2) goto q1:
        else want2 = true;
    }
    seção crítica
```

```
    gate 2 = 0;
    want2 = false;
}
```

Starvation - processo 2 espera indefinitivamente

trocar por for all others await

want [j] == false