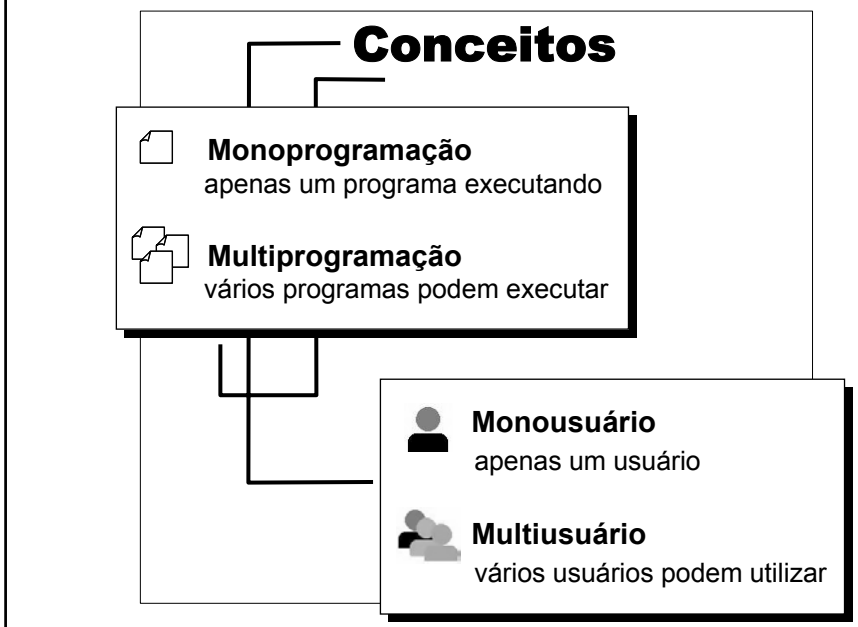
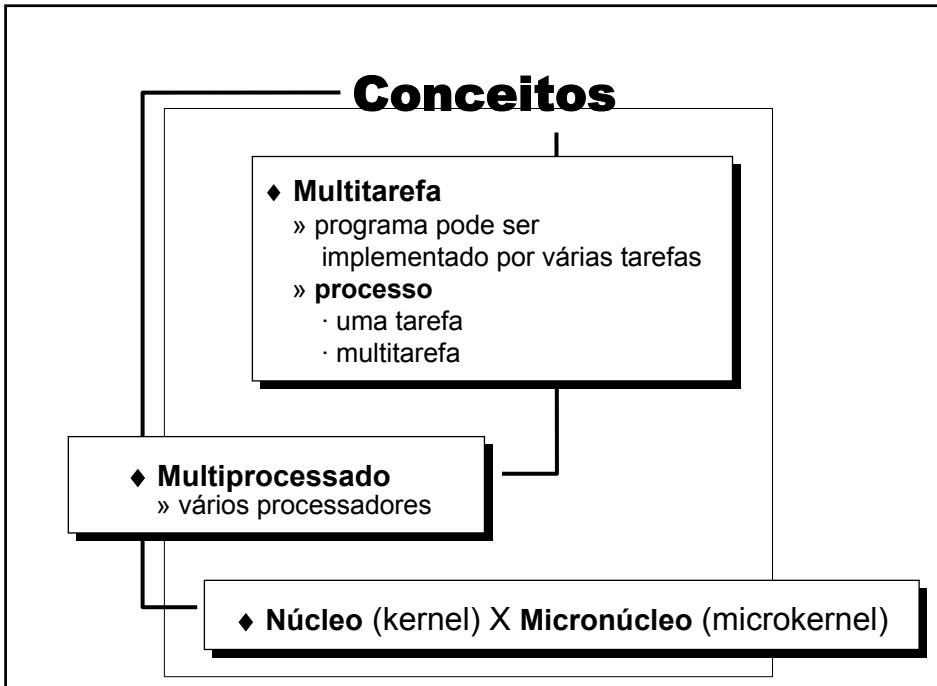
Sistemas Concorrentes e Distribuídos

Introdução

Roteiro

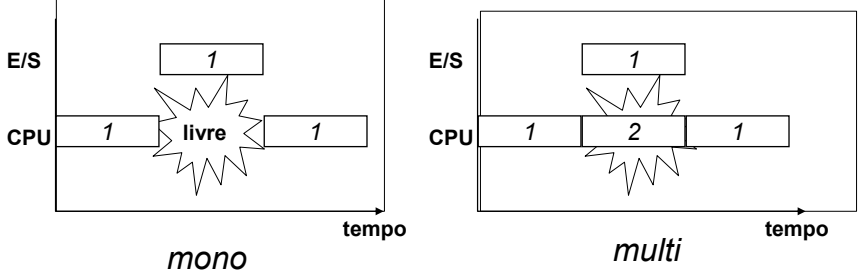
- Conceitos básicos de concorrência
- Semáforos
- Problemas clássicos
- Verificação
  - Redes de Petri
- Monitores
- Troca de Mensagens
- Verificação
- RPC / RMI
- Tolerância a Falhas


Conceitos
Monoprogramação
apenas um programa executando
Multiprogramação
vários programas podem executar
Monousuário
apenas um usuário
Multiusuário
vários usuários podem utilizar


Conceitos
♦ Multitarefa
» programa pode ser implementado por várias tarefas
» processo
· uma tarefa
· multitarefa
♦ Multiprocessado
» vários processadores
♦ Núcleo (kernel) X Micronúcleo (microkernel)

Sistemas Multiprogramados

- **concorrência**
  - princípio do intercalamento
  - maior utilização da CPU



Programação Concorrente

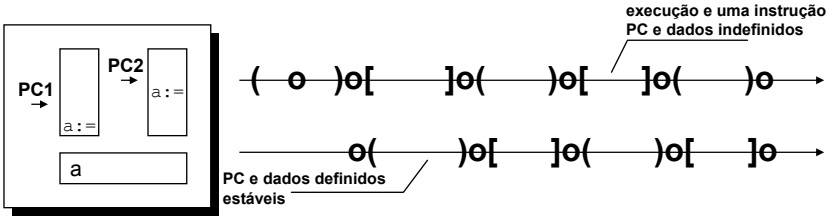
primeiros conceitos

- **explorar o paralelismo**
- **várias aplicações executadas compartilhando a mesma CPU**
- **aplicações estruturadas de forma que seus módulos podem ser executados concorrentemente**
  - **sem interação**
  - **com interação (compartilhando recursos)**
- **o SO pode (e precisa) utilizar esta técnica internamente**
  - **"o SO pode ser visto como um grande monitor..."**

Programação Concorrente

Hipóteses

- **teórica**
  - **programas distribuídos**
  - **funcionam em qualquer plataforma**
  - **independe da velocidade de execução**
- **intercalamento no tempo**
  - **vários cenários de execução possíveis**
  - **velocidade do PC de cada processo é independente**
  - **não determinística**

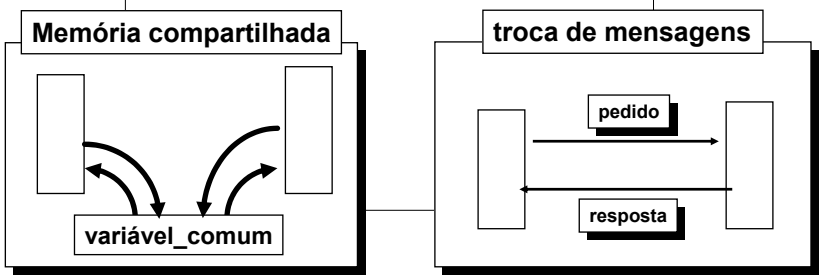


Especificação de Concorrência em Programas

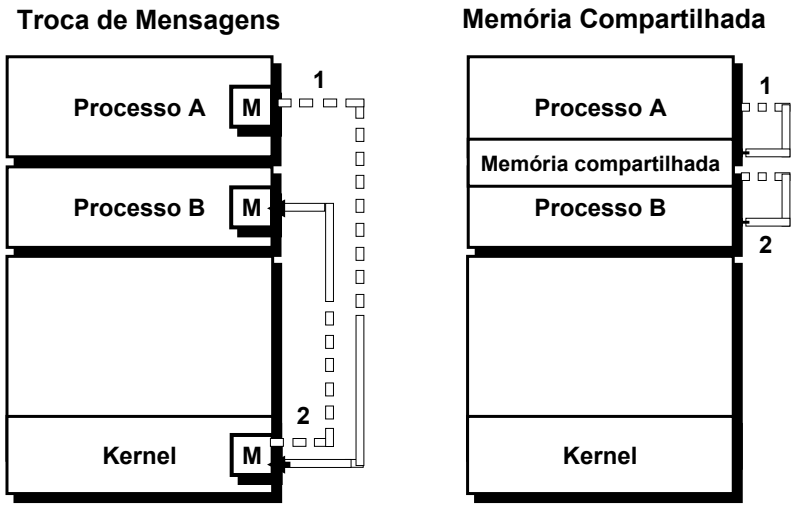
- **Linguagens:**
  - **Pascal Concorrente: cobegin - coend**
  - **OOCAM**
  - **ADA**
- **Unix:**
  - **fork**
  - **wait**
- **Mecanismos para compartilhamento de Recursos**
  - **memória compartilhada**
  - **troca de mensagens**

Interação entre Processos

- **Processos (que compõem um mesmo programa ou não) podem precisar cooperar entre si**
- **Mecanismos para compartilhamento de Recursos**



Communication Models



Problemas de Comunicação e Exclusão Mútua

- **comunicação entre processos**
  - **IPC - Inter Process Communication**
  - **implementação do compartilhamento de recursos**
- **problema: inconsistência nos dados compartilhados**
  - **solução: exclusão mútua e sincronização**
  - **garantia de que algumas condições serão respeitadas na comunicação entre processos**
- **outros problemas**
  - **deadlock**
  - **livelock**
  - **starvation**
  - **injustiça**

Exclusão Mútua

- **Requisitos**
  - **apenas um processo pode acessar a região crítica no acesso a recursos compartilhados**
  - **a entrada ou saída de processos *competindo* pelo mesmo recursos não deve influenciar outros processos**
  - **ausência de deadlock ou starvation**
  - **nada é assumido em relação à velocidade de execução dos processos nem ao número dos processos**
  - **um processo não conhece implicitamente o estado de outro processo**
    - » **daí a necessidade de se garantir o sincronismo e a exclusão mútua**

Região Crítica

- **uma abstração**
- **"área de segurança" para acesso à recursos compartilhados**
- **um processo desejando entrar na RC**
  - deve ser garantido ao processo a entrada na RC
  - se a RC estiver livre, deve poder fazê-lo sem atrasos
- **um processo só pode ficar na RC durante um período finito de tempo**

```
program exclusão_mutua;
const n = ...; { número de processos }

  procedure P (i : integer);
  Begin
    repeat
      <código restante>
      entra_reg_crítica (R);
       <código para a região crítica R>
      sai_reg_critica (R);
       <código restante>
    forever
  End;

Begin
    cobegin
       P (1);
       P (2);
       ...
       P (n);
    coend
End.
```


Sincronização

- **Velocidade de execução dos processos é diferente**
  - **uso de buffers**
  - **lê o que ainda não foi escrito**
  - **inconsistência em banco de dados**
  - **processos tem "visão" diferente de variáveis**
    - » **problemas semelhantes ao de cache**

Exclusão Mútua - Soluções por Software

- **algoritmo de Dekker**
  - **Djikstra**
    - » **propôs série de soluções evolutivas**
    - » **cada uma evidencia bugs comuns em programas concorrentes**
      - **1 - uma variável, buzy waiting**
      - **2 - duas variáveis**
      - **3 - 3 variáveis**
      - **4 - correta**
- **algoritmo de Peterson**
  - **garante exclusão mútua**
  - **garante justiça para 2 processos**

Primeira Tentativa

- espera ocupada (busy-waiting) => *while*
- alternância explícita dos processos
- velocidade ditada pelo mais lento
- falhas na RC ?

variável global compartilhada => **turn**: 0..1;

Processo 0

```
:
:
while turn <> 0 do { nothing };
< região crítica >
turn := 1;
:
:
```

Processo 1

```
:
:
while turn <> 1 do { nothing };
< região crítica >
turn := 0;
:
:
```

TRACE – Primeira Tentativa

Processo 0 → P0 Processo 1 → P1 t → tempo
— → processo parado × → processo executando

t	P0	P1	turn	Observação
t0	—	—	—	inicialização
t1	—	—	0	—
t2	×	—	0	inicia P0
t3	—	×	0	inicia P1
t4	—	×	0	teste while ok
t5	—	×	0	{nothing}
t6	×	—	0	teste while falha
t7	®	—	0	região crítica de P0
t8	®	×	0	teste while ok
t9	×	—	1	turn:=1
t10	×	—	1	final
t11	×	—	1	teste while ok
t12	×	—	1	{nothing}
t13	—	×	1	{nothing}
t14	—	×	1	teste while falha
t15	—	®	1	região crítica de P1

Segunda Tentativa

- cada processo tem sua própria chave para a RC
- cada processo vê o quadro de avisos do outro mas não pode alterá-lo
- ~~não existe bloqueio se outro processo falha fora da RC~~ (o mesmo não é garantido se este falha dentro da RC)
- existe falha grave no algoritmo (não garante exclusão mútua)

variável global compartilhada => var **flag**: array [0..1] of boolean;

Processo 0

```
:
while flag[ 1 ]  do { nothing };
flag[ 0 ] := true;
< região crítica >
flag[ 0 ] := false;
:
```

Processo 1

```
:
while flag[ 0 ]  do { nothing };
flag[ 1 ] := true;
< região crítica >
flag[ 1 ] := false;
:
```

TRACE – Segunda Tentativa

Processo 0 → P0 Processo 1 → P1 t → tempo
— → processo parado × → processo executando

t	P0	P1	flag[0]	flag[1]	Observação
t0	—	—	false	true	inicialização
t1	×	—	false	false	inicia P0
t2	—	×	false	false	inicia P1
t3	×	—	false	false	while flag[1]==true ?
t4	—	×	false	false	while flag[0]==true ?
t5	×	—	true	false	flag[0]:=true
t6	—	×	true	true	flag[1]:=true
t7	®	—	true	true	região crítica de P0
t8	®	®	true	true	região crítica de P1

Quebra da exclusão mútua

Terceira Tentativa

- em relação à segunda tentativa, apenas uma mudança no código
- garante exclusão mútua
- problemas de deadlock (os dois setam o respectivos flags para 1)
- cada processo pode setar o valor de seu flag sem saber da condição do outro

Processo 0

```
:
flag[ 0 ] := true;
while flag[ 1 ]  do { nothing };
< região crítica >
flag[ 0 ] := false;
:
```

Processo 1

```
:
flag[ 1 ] := true;
while flag[ 0 ]  do { nothing };
< região crítica >
flag[ 1 ] := false;
:
```

TRACE – Terceira Tentativa

Processo 0 → P0 Processo 1 → P1 t → tempo
— → processo parado × → processo executando

t	P0	P1	flag[0]	flag[1]	Observação
t0	—	—	false	false	inicialização
t1	×	—	false	false	inicia P0
t2	—	×	false	false	inicia P1
t3	×	—	true	false	flag[1]:=true
t4	—	×	true	true	flag[1]:=true
t5	×	—	true	true	while flag[1]==true ?
t6	—	×	true	true	while flag[0]==true ?
t7	×	—	true	true	{nothing}
t8	—	×	true	true	{nothing}
t9	×	—	true	true	while flag[1]==true ?
t10	—	×	true	true	while flag[0]==true ?

Espera Ocupada
Deadlock

Quarta Tentativa

- processos agora setam seu flag para indicar a intenção
- pode haver livelock (tracing abaixo) “mutual courtesy”

Process 0

```
.
flag[0] := true;
while flag[1] do
   begin
      flag[0] := false;
      <delay for a short time>;
      flag[0] := true;
   end;
< critical section >;
flag[0] := false;
.
```

Process 1

```
.
flag[1] := true;
while flag[0] do
   begin
      flag[1] := false;
      <delay for a short time>;
      flag[1] := true;
   end;
< critical section >;
flag[1] := false;
```

PO sets flag [0] to true
P1 sets flag [1] to true
P0 checks flag [1]
P1 checks flag [0]
P0 sets flag [0] to false
P1 sets flag [1] to false
P0 sets flag [0] to true
P1 sets flag [1] to true

flag[0] = TRUE
flag[1] = TRUE

Solução Correta

- **uso da variável turn**
- **as variáveis flag ainda são usadas**
- **algoritmo original de Dekker**
- **algoritmo complexo e de prova difícil**

Algoritmo de Dekker

```
var flag: array [0 .. 1] of boolean;
    turn: 0 .. 1;
```

```
procedure P0;
begin
 repeat
  flag [0] := true;
  while flag [1] do
    if turn = 1 then
       begin
         flag [0] := false;
         while turn = 1 do {nothing};
         flag [0] := true
       end;
  < região crítica >;
  turn := 1;
  flag [0] := false;
  < restante >
 forever
end;
```

```
procedure P1;
begin
 repeat
 flag [1] := true;
 while flag [0] do
   if turn = 0 then
      begin
         flag [1] := false;
         while turn = 0 do {nothing};
         flag [1] := true
      end;
  < região crítica >;
  turn := 0;
  flag [1] := false;
  < restante >
 forever
end;
```

```
Begin
  flag [0] := false;
  flag [1] := false;
  turn := 1;
  parbegin
    P0; P1
  parend
end.
```

Algoritmo de Petterson

- garante exclusão mútua e justiça (para 2 processos ?)
- também usa flags (mutex) e turn (conflitos)

```
var  flag array [0 ..1] of boolean;
     turn:0..1;
```

```
procedure P0;
begin
 repeat
  flag [0] := true;
  turn := 1;
  while flag [1] and turn = 1 do
                        {nothing};
  < região crítica >;
  flag [0] := false;
  < restante >
 forever
end;
```

```
procedure P1;
begin
 repeat
 flag [1] := true;
 turn := 0;
 while flag [0] and turn = 0 do
                          {nothing};
 < região crítica >;
 flag [1] := false;
 < restante >
 forever
end;
```

```
begin
  flag [0] := false;
  flag [1] := false;
  turn := 1;
  parbegin P0; P1 parend
end.
```


Sincronização - Soluções por Hardware

- **desabilitar interrupções**
- **instruções implementadas por hardware**
  - **test-and-set (atômicas)**
  - **exchange**
  - **protocolo de exclusão mútua**
    - » **ex.:**
      - **repeat { nothing } until testeset (variável);**
    - » **ex.:**
      - **repeat exchange (key_i, bolt) until key_i = 0;**
  - **simples, vários processos, várias RCs**
  - **espera ocupada, starvation, deadlock**