UFRJ – IM - DCC

Sistemas Operacionais I

Unidade II – Concorrência

Organização da Unidade

- **Processos**
- **Threads**
- **Concorrência**
  - Princípios da Concorrência
  - Exclusão Mútua – Conceituação
  - Estratégias de Implementação – Semáforos, Monitores e Mensagens
  - Problemas Clássicos
- ***Deadlock* e *Starvation***

O que é Concorrência

Consiste, num ambiente que admite a execução paralela ou concorrente de vários processos, dos eventos em que mais de um processo requisita (concorre) a posse de um mesmo recurso.

Como o recurso não pode ser distribuído para mais de um processo ao mesmo tempo, providências devem ser tomadas de forma a evitar inconsistências que venham a prejudicar a execução dos processos.

Contextos que levam à Concorrência

- Múltiplas Aplicações
  - Multiprogramação surgiu para permitir que diferentes aplicações compartilhem de forma concorrente os recursos do computador.
- Aplicações Estruturadas
  - Na programação estruturada uma aplicação pode ser construída como um conjunto de sub-processos.
- Estrutura do Sistema Operacional
  - O próprio sistema operacional é implementado como um conjunto de processos concorrentes.

Multiprogramação - Problemas Inerentes

- A velocidade de execução de um processo não pode ser prevista, ela depende das atividades dos outros processos, da forma como o SO trata as interrupções e das estratégias de escalonamento adotadas.
- O compartilhamento de variáveis globais é cheio de perigos.
- É difícil para o SO administrar de forma ótima a alocação de recursos. (Exemplo: um processo pode requerer um canal de I/O e ser suspenso pelo SO antes de usá-lo)
- Como a repetição do contexto é impraticável, pode tornar-se muito mais difícil localizar um erro de execução do programa.

Exemplo de Compartilhamento

```
procedure echo;
var out, in: char;
begin
      input(in, keyboard);
      out:= in;
      output(out, display);
end.
```

Processo P1	Processo P2
.	.
input(in, keyboard);	.
.	input(in, keyboard);
out:= in;	.
.	out:= in;
output(out, display);	.
.	output(out, display);

Responsabilidades do SO

- O SO precisa:
  - Monitorar as atividades dos diferentes processos em execução.
  - Alocar e desalocar recursos para os diferentes processos
    - Tempo de CPU
    - Memória
    - Arquivos
    - Dispositivos de E/S, etc
  - Proteger os dados e os recursos de cada processo contra interferências não intencionadas de outros processos
  - Fazer com que os resultados de um processo independam da velocidade relativa com que o mesmo é executado.

Relação entre Processos

Processos : independentes ou cooperativos

✓Processos independentes
Não afetam e nem são afetados pelos demais processos

✓Processos cooperativos
Afetam e são afetados pelos demais processos do grupo

E em relação ao compartilhamento ???

Comunicação entre processos

- ***IPC – Inter-process Communication***
  - Memória compartilhada
  - Troca de mensagem
    - *Comunicação direta*
    - *Comunicação indireta*

Interação entre Processos – Quadro Geral

Grau de Percepção	Relacionamento	Influência	Problemas
Desconhecimento	Competição	- resultados independentes - tempo de processamento pode ser afetado	- Exclusão Mútua - Deadlock (recursos renováveis) - Starvation
Conhecimento Indireto (compartilham algum objeto)	Cooperação por Compartilhamento	- resultado de um pode depender de informação do outro - tempo de processamento pode ser afetado	- Exclusão Mútua - Deadlock (recursos renováveis) - Starvation - Coerência de Dados
Conhecimento Direto (primitiva de comunicação)	Cooperação por Comunicação	- resultado de um pode depender de informação do outro - tempo de processamento pode ser afetado	- Deadlock (recursos consumíveis) - Starvation

Relação entre processos - Resumo

Competição
- ✓Processos independentes
- ✓Compartilhamento de recursos

Cooperação

Cooperação por compartilhamento
- ✓dependência indireta

Cooperação por comunicação
- ✓ dependência direta

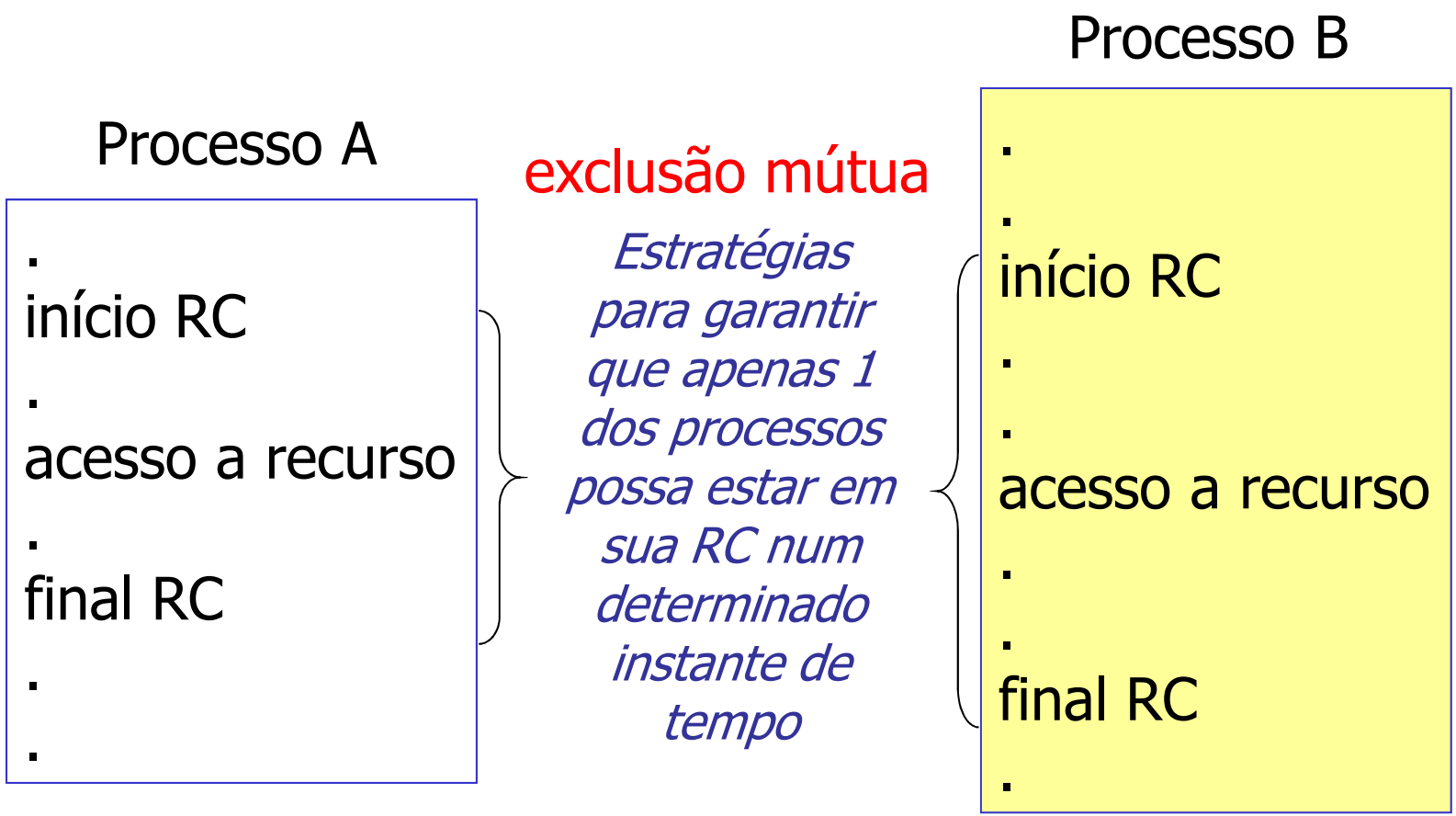
Alguns Conceitos

- **Condição de corrida**
  Conjunto de eventos que levam a resultados não determinísticos.
- **Região Crítica**
  Parte do código que implementa o acesso a um recurso compartilhado.
- **Exclusão mútua**
  Mecanismo que impede o uso simultâneo de um recurso compartilhado.
- **Sincronização**
  Mecanismo que possibilita a manutenção da ordem (seqüência) na execução de determinadas operações a serem realizadas por processos distintos.

Requisitos para Exclusão mútua

1. *Somente um processo por vez pode executar uma região crítica*
2. *Um processo interrompido fora da RC não pode impedir que outro processo a acesse*
3. *Não pode haver nem deadlock nem starvation*
4. *Quando não houver processo executando uma RC, qualquer processo que solicitar acesso deve obtê-lo imediatamente*
5. *Um processo deve permanecer na RC por tempo finito*
6. *Nada pode ser assumido sobre a velocidade relativa e dos processos nem o número de processadores*

Exclusão Mútua - Implementação



Exclusão Mútua – Formas de Implementação

Abordagem por Software (sem suporte)

Algoritmos de espera ocupada

Abordagens com suporte de Hardware

- Instruções atômicas
- Desabilitar interrupções

Abordagens com suporte do Sistema Operacional

- Semáforos
- Monitores
- Troca de mensagens

Abordagem por Software

Algoritmo de Dekker

var turn: 0 .. 1; (variável global)

Processo 0

```
.
while turn ≠ 0 do { };
<RC>
turn := 1;
.
```

Processo 1

```
.
while turn ≠ 1 do { };
<RC>
turn := 0;
.
```

Algum problema de execução??

Algoritmo de Dekker - Problemas

Garante a exclusão mútua, porém gera dois problemas:

- Processos se alternam no uso de suas respectivas RCs, o tempo de execução será ditado pelo processo mais lento.
- Se um dos processos falhar (abortar por exemplo) o outro jamais poderá entrar em sua RC novamente.

Abordagem por Software

Algoritmo de Dekker – 2ª abordagem

var flag: array [0 .. 1] of boolean; (variável global)

Processo 0

```
.
while flag[1] do { };
flag[0] := true;
<RC>
flag[0] := false;
.
```

Processo 1

```
.
while flag[0] do { };
 flag[1] := true;
<RC>
 flag[1] := false;
.
```

Algum problema de execução??

Algoritmo de Dekker - Problemas

- Corrigiu: os processos não mais se revezam no acesso às suas RCs
- Corrigiu em parte: se um processo falha fora da sua RC ele não bloqueia o outro processo.

Grave: Não garante mais a exclusão mútua

Abordagem por Software

Algoritmo de Dekker – 3ª abordagem

var flag: array [0 .. 1] of boolean; (variável global)

Processo 0

```
.
flag[0] := true;
while flag[1] do { };
<RC>
flag[0] := false;
.
```

Processo 1

```
.
 flag[1] := true;
 while flag[0] do { };
<RC>
 flag[1] := false;
.
```

Algum problema de execução?? ***Pode causar Deadlock***

Abordagem por Software

Algoritmo de Dekker – 4ª abordagem

var flag: array [0 .. 1] of boolean; (variável global)

Processo 0

```
.
flag[0] := true;
while flag[1] do
{ flag[0] := false;
<delay randômico>
flag[0] := true;}
<RC>
flag[0] := false;
.
```

Processo 1

```
.
flag[1] := true;
while flag[0] do
{flag[1] := false;
 <delay randômico>
 flag[1] := true;}
 <RC>
 flag[1] := false;
.
```

Algum problema de execução??

Algoritmo de Dekker - Problemas

P0 seta flag[0] para false
P1 seta flag[1] para false
P0 verifica flag[1]
P1 verifica flag[0]
P0 seta flag[0] para false
P1 seta flag[1] para false
P0 seta flag[0] para true
P1 seta flag[1] para true

Pode se repetir indefinidamente

Starvation

Abordagem por Software

Algoritmo de Dekker – Versão Correta

```
var flag: array [0..1] of boolean;
    turn: 0 .. 1;
procedure P0;
{repeat
      flag[0] := true;
      while flag[1] do if turn = 1 then
            {flag[0] := false;
              while turn = 1 do { };
              flag[0] := true;}
      <RC>
      turn := 1;
      flag[0] := false;
      . . .
      forever;
}
```

```
procedure P1;
{repeat
      flag[1] := true;
      while flag[0] do if turn = 0
      then
            {flag[1] := false;
              while turn = 0 do { };
              flag[1] := true;}
      <RC>
      turn := 0;
      flag[1] := false;
      . . .
      forever;
}
```

Desabilitar Interrupções

- Resultado: Impede que o processo seja interrompido

EXEMPLO: CLI
<RC>
STI

Restrições
- *Não funciona em sistemas multiprocessados*
- *Problemas com a integridade do sistema*

Suporte de Hardware

Instruções Atômicas – T&S

- Conjunto de Instruções especiais
- Execução atômica
- Simples de utilizar
- Utiliza espera ocupada
- Possibilidade de starvation;
- seleção arbitrária
- Exclusividade no uso do barramento para multiprocessadores

```
ENTRA_RC
        TSL  RX,LOCK
        CMP RX , #0
        JNE  ENTRA_RC
          :
          :
          :
SAI_RC
        MOV LOCK , #0
        RET
```

Suporte de Hardware

Como função

```
function testset (var i: integer): boolean;
{
    if i = 0 then {
      i := 1;
      testset := true;
    }
    else testset := false;
}.
```

Como rotina

```
procedure exchange (var r: register; var m: memory);
var temp;
{
    temp := m;
    m := r;
    r := temp;
}.
```

Suporte de Hardware

```
program exclusao_mutua_exchange;
const        n = ... (número de processos);
var          bolt: integer;
procedure P(i: integer);
{repeat
        key_i := 1;
        repeat exchange(key_i, bolt) until key_i = 0;
        <RC>
        exchange(key_i, bolt)
        .
        forever;
}
```

```
program exclusao_mutua_tsl;
const        n = ... (número de processos);
var          bolt: integer;
procedure P(i: integer);
{repeat
        repeat { } until testset(bolt);
        <RC>
        .
        bolt = 0;
        forever;
}
```

```
begin (*main program*)
  bolt = 0;
  parbegin
    P(1);
    P(2);
    .....
     P(n);
  parend
}.
```

(*) em ambos casos bolt inicia com 0

Abordagem com Suporte do SO

Semáforos

- Solução proposta por Dijkstra em 1965
- Principio básico:

  um processo é suspenso enquanto não obtém permissão para executar uma RC e é automaticamente "acordado" através de um sinal;

- Para liberar um semáforo *"s"* o processo deve executar uma primitiva signal(*s*) e para obter acesso a RC via o semáforo *"s"* o processo deve executar um primitiva wait(*s*).

Abordagem com Suporte do SO

Semáforos - Primitivas

- O semáforo deve ser inicializado com um valor não negativo.

- A operação wait decrementa o semáforo; se o valor ficar negativo o processo é bloqueado.

- A operação signal incrementa o semáforo; se o valor não ficar positivo o processo bloqueado pela operação wait é desbloqueado.

Abordagem com Suporte do SO

Semáforos - Implementação

```
type  semaphore =  record
                       count: integer;
                       queue: list of process
                   end;
var s: semaphore;
```

Rotina wait()

```
wait(s):
        s.count := s.count - 1;
        if s.count < 0
           then begin
                place this process in s.queue;
                block this process
           end;
```

Rotina signal()

```
signal(s):
        s.count := s.count + 1;
        if s.count ≤ 0
           then begin
                remove a process P from s.queue;
                place process P on ready list
           end;
```

Uso de Semáforo - Exemplo

Produtor /Consumidor

- Produtores gerando dados para um buffer
- Consumidores retirando dados de um buffer
- Somente um processo pode ter acesso ao buffer
- 2 semáforos: controle de acesso ao buffer e sincronização

Qual o valor inicial de cada semáforo?

Produtor /Consumidor

Caso 1: Buffer infinito

```
Produtor (i) {

      while (T) {
          produz_item()
          wait(s)
            adiciona_item()
          signal(s)
          signal(n)
      }
}
```

```
Consumidor(i) {

while (T) {
          wait(n)
          wait(s)
            remove_item()
          signal(s)
          consome_item()
      }
}
```

s – controle de acesso / n - sincronização

Produtor /Consumidor

Caso 2: Buffer finito

```
Produtor (i) {
        while (T) {
           produz_item()
           wait(v)
           wait(s)
             adiciona_item()
           signal(s)
           signal(n)
        }
}
```

```
Consumidor(i) {
 while (T) {
           wait(n)
           wait(s)
             remove_item()
           signal(s)
           signal(v)
           consome_item()
           }
}
```

s – acesso / n – itens produzidos / v – tamanho do buffer

Abordagem com Suporte do SO

```
type binary semaphore = record
            value: (0, 1);
            queue: list of processes
            end;
var s: binary semaphore;
```

```
waitB(s)
            if s.value = 1 then
                  s.value := 0;
            else
                  place the process in s.queue
                  block the process
            end;
end;
```

```
signalB(s)
            if s.queue = empty then
                  s.value := 1;
            else
                  remove a process from s.queue
                  place the process on Ready queue
            end;
end.
```

Semáforos Binários

Exemplo: produtor/consumidor

Buffer infinito/Semáforos Binários

n =0 (contador) s=1(mutex) a=0(atraso)

```
Produtor (i) {

        while (T) {
            produz_item()
            wait(s)
  RC        adiciona_item()
            n=n+1
            signal(s)
            se n=1 entao signal(a)
        }
}
```

```
Consumidor(i) {

wait(a)
while (T) {
            wait(s)
               remove_item()
  RC           n=n-1
            signal(s)
            consome_item()
            se n = 0 entao wait(a)
        }
}
```

Execução do exemplo

Problema

Rotina	Ação	N	A
P	RC	1	1
C	wait(a)	1	0
C	RC	0	0
P	RC	1	1
C	se n=0	1	1
C	RC	0	1
C	se n= 0	0	0
C	RC	-1	0

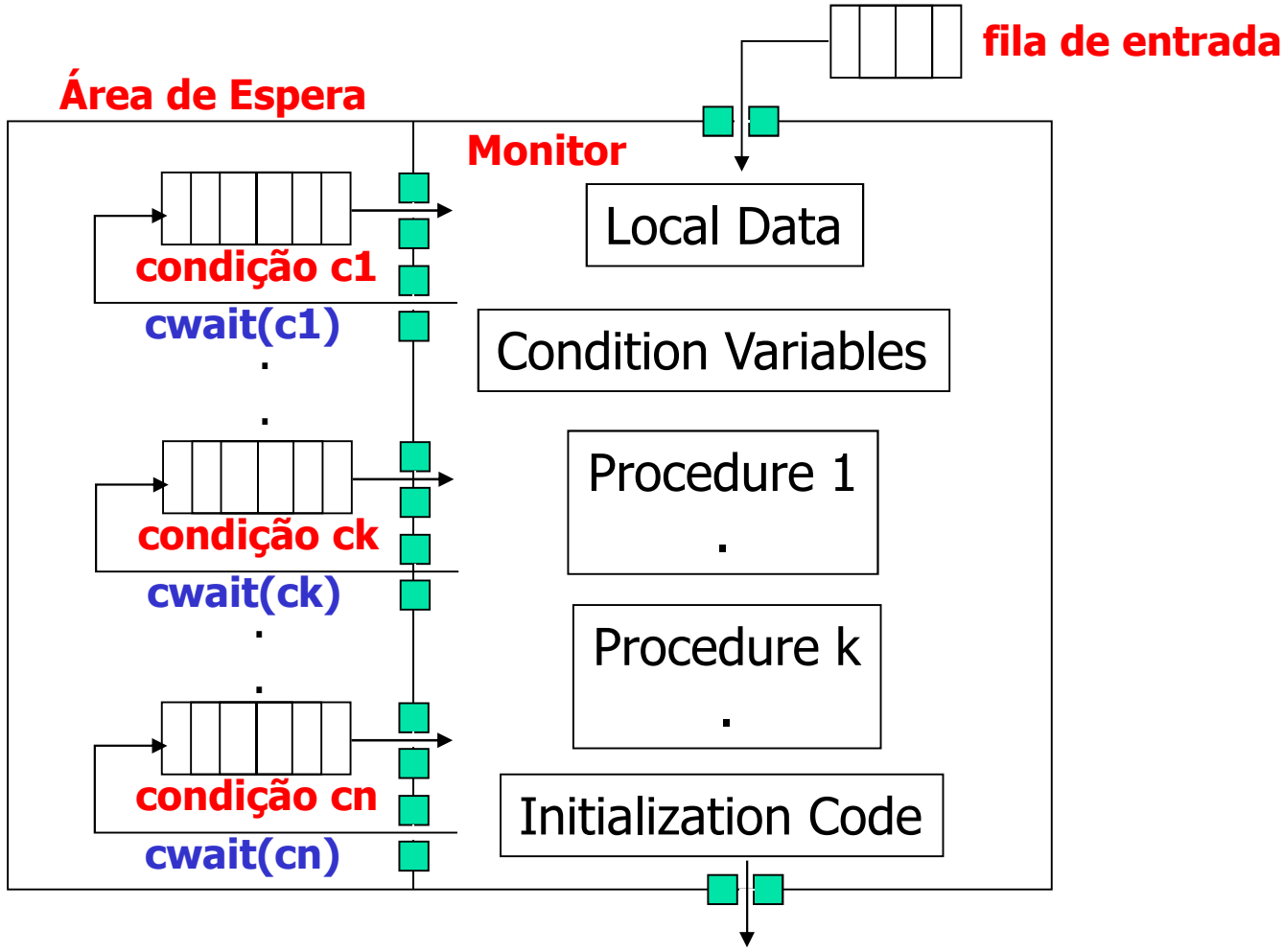
Abordagem com Suporte do SO

Monitores

- Estrutura de linguagem
  - Pascal – concorrente
  - Modula 2
- O monitor é composto de várias rotinas cujas variáveis somente são acessadas dentro do monitor
- Um processo entra no monitor chamando uma das suas rotinas
- Somente um processo entra no monitor de cada vez

Abordagem com Suporte do SO

Estrutura de um Monitor



Abordagem com Suporte do SO

Troca de mensagem

- Também utilizada em sistemas distribuídos
- Utiliza as primitivas *send* e *receive*
- Estas primitivas podem ser bloqueantes ou não bloqueantes

Abordagem com Suporte do SO

Troca de mensagem

```
program        mutualexclusion;
const n = . . . ; (*number of processes*);
procedure P(i: integer);
var msg: message;
begin
    repeat
        receive (mutex, msg);
        < critical section >;
        send (mutex, msg);
        < remainder >
    forever
end;
```

```
begin (* main program *)
    create_mailbox (mutex);
    send (mutex, null);
    parbegin
        P(1);
        P(2);
        . . .
        P(n)
    parend
end.
```

Abordagem com Suporte do SO

Características do Sistema de Mensagens

- Sincronização
  - Send
    - Blocking
    - Nonblocking
  - Receive
    - Blocking
    - Nonblocking
    - Test for Arrival

- Endereçamento
  - Direto
    - Send
    - Receive
      - Explicito
      - Implícito
  - Indireto
    - Estático
    - Dinâmico

(*) endereçamento indireto desacopla os processos

Abordagem com Suporte do SO