Aula 4

Comunicação e Sincronização de Processos

Prof. Sérgio Campos / Prof. Marcus Rocha {scampos,mvrocha}@dcc.ufmg.br

Comunicação e Sincronização de Processos

- Cooperação
- Variáveis compartilhadas
- Operações atômicas
- Exclusão mútua
- Requisitos para exclusão mútua

Referências:

- Capítulo 4: 4.4
- Capítulo 6: 6.1

Porque que compartilhar?

Compartilhamento de recursos:

- Um computador (impressora, etc), muitos usuários

Multiprocessadores:

- Resolve o problema mais rápido se dividir o problema entre os processadores

Modularidade:

- Divida o problema em problemas menores (ex: escalonador, comunicação entre processos)

Sistemas distribuídos:

- Internet

Cooperação – *race conditions* – corrida crítica

Processos podem se comunicar através do compartilhamento de variáveis

Quais problemas podem ocorrer quando dois ou mais processos compartilham variáveis?

Cooperação – *race conditions* – corrida crítica

Considere 2 processos que compartilham as variáveis A e B:

P_1
A = 1

P_2
B = 2

Qual será o resultado final? A ordem de execução dos processos tem importância?

Qual é o resultado final?

Cooperação – *race conditions* – corrida crítica

Considere 2 processos que compartilham as variáveis A e B:

P_1	P_2	Qual será o resultado final? A ordem de execução dos processos tem importância?
A = 1	B = 2	

Agora considere:

P_1	P_2	Qual é o resultado final?
A = B + 1	B = 2 * B	

Cooperação – *race conditions* – corrida crítica

Considere 2 processos que compartilham as variáveis A e B:

P_1	P_2
A = 1	B = 2

Qual será o resultado final? A ordem de execução dos processos tem importância?

Agora considere:

P_1	P_2
A = B + 1	B = 2 * B

Qual é o resultado final?

E no seguinte caso?

P_1	P_2
A = 1	A = 2

Qual é o resultado final?

E em um computador com múltiplos processadores?

Operações Atômicas

A fim de evitar *race conditions* o conceito de **operações atômicas** é introduzido:

Operações atômicas são operações que não podem ser interrompidas:

- Não é possível ver as “partes” de uma operação atômica, mas apenas seu efeito final. Ou seja, não é possível ver uma “operação em progresso”

Atômicas	**Não-Atômicas**
tocar a campainha	encher um copo de água
desligar a luz	caminhar até a porta

Operações Atômicas

Operações atômicas são relevantes em outras áreas além dos Sistemas Operacionais:

- Elas são a base para **transações atômicas** que, por sua vez, formam uma base para uma área denominada **Processamento de Transações**.

- Esta área trata de problemas de coordenação de acessos múltiplos e concorrentes a bancos de dados:
  - Bancos eletrônicos são uma das aplicações importantes desta área

Operações Atômicas

En geral, o *hardware* provê algumas operações atômicas:

- Se o hardware não fornecer as operações atômicas, como é possível implementá-las em um uniprocessador? E em multiprocessador?

Sincronização

Sincronização é uma forma de garantir o funcionamento correto de processos cooperantes:

- Operações atômicas são usadas

Sincronização

O problema do espaço na geladeira

Hora	Pessoa A	Pessoa B
6:00	Olha a gel.: sem leite	...
6:05	Sai para a padaria	...
6:10	Chega na padaria	Olha a gel.: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Em casa: guarda leite	Chega na padaria
6:25	...	Sai da padaria
6:30	...	Chega em casa: Ops!

O que houve de errado?

- Problema: falta de **comunicação**! E vocês achavam que isto era papo de psicólogo...

Sincronização

O problema anterior era causado porque uma pessoa não sabia o que a outra estava fazendo.

Uma solução para o problema envolve dois novos conceitos:

- **Exclusão mútua:** apenas um processo pode fazer alguma coisa em determinado momento.
  - Exemplo: apenas uma pessoa pode sair para comprar leite em qualquer momento
- **Seção crítica:** uma seção de código na qual apenas um processo pode executar de cada vez,
  - O objetivo é tornar **atômico** o conjunto de operações
  - Exemplo: comprar leite

Exclusão Mútua

Existem várias maneiras de se obter exclusão mútua:

A maioria envolve **trancamento** (*locking*):

- Evitar que alguém faça alguma coisa em determinado momento.

  Exemplo: deixar um aviso na porta da geladeira.

Três regras devem ser satisfeitas para o trancamento funcionar:

Regra	**Exemplo da geladeira**
1. Trancar antes de utilizar	Deixar aviso
2. Destrancar quando terminar	Retirar o aviso
3. Esperar se estiver trancado	Não sai para comprar se houver aviso

Computadores & Leite

Primeira tentativa de resolver o *Problema do Espaço na Geladeira*:

Processos A e B

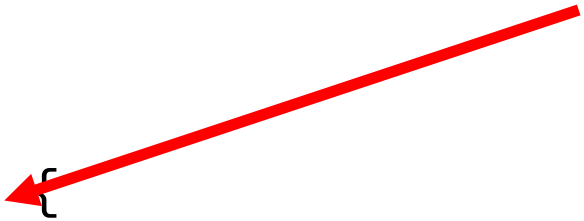
```
if (SemLeite) {
   if (SemAviso) {
      Deixa Aviso;
      Compra Leite;
      Remove Aviso;
   }
}
```

Esta “solução” funciona?

Computadores & Leite

Não, por causa da **troca de contexto**.

Processos A e B



```
if (SemLeite) {
   if (SemAviso) {
      Deixa Aviso;
      Compra Leite;
      Remove Aviso;
   }
}
```

A “solução” piora o problema! Agora, falha só de vez em quando, ou seja, a depuração fica muito mais difícil.

- **Heisenbug**
- Não importa se é raro, na prática vai acontecer nos primeiros 5 minutos. A não ser quando você estiver querendo que aconteça!

Computadores & Leite

Segunda tentativa de resolver o *Problema do Espaço na Geladeira* – mudar o significado do aviso:

Processo A

```
if (SemAviso) {
   if (SemLeite) {
      Compra Leite;
   }
   Deixa Aviso;
}
```

Processo B

```
if (Aviso) {
   if (SemLeite) {
      Compra Leite;
   }
   Remove Aviso;
}
```

Funciona? Porque?

Computadores & Leite

Que tal o seguinte argumento?

- Somente A deixa um aviso, e somente se já não existe um aviso;
- Somente B remove um aviso, e somente se houver um aviso;
- Portanto, ou existe um aviso, ou nenhum;
- Se houver aviso, B compra leite;
- Se não houver aviso, A compra leite;
- Portando, apenas uma pessoa (processo) vai comprar leite.

Certo?

Computadores & Leite

Suponha que B saia de férias:

- A vai comprar leite uma vez e não vai comprar mais até que B retorne.

- Portanto, esta solução não é boa; em particular, ela pode levar a uma **inanição** (*starvation*).

(Tá bom, trapaça. Eu nunca disse nada sobre inanição. Mas não é importante?)

Computadores & Leite

Terceira tentativa – usar dois avisos diferentes:

Processo A

```
Deixa AvisoA;
while (AvisoB);
if (SemLeite) {
   Compra Leite;
}
Remove AvisoA;
```

Processo B

```
Deixa AvisoB;
if (SemAvisoA) {
   if (SemLeite) {
      Compra Leite;
   }
}
Remove AvisoB;
```

Funciona?

Computadores & Leite

SIM!

Processo A

```
Deixa AvisoA;
while (AvisoB);
if (SemLeite) {
    Compra Leite;
}
Remove AvisoA;
```

Processo B

```
Deixa AvisoB;
if (SemAvisoA) {
    if (SemLeite) {
        Compra Leite;
    }
}
Remove AvisoB;
```

- Se SemAvisoA B pode comprar porque A ainda não começou.
- Se AvisoA, A está comprando ou esperando até que B desista. B pode desistir.
- Se SemAvisoB A pode comprar.
- Se AvisoB, não se sabe:
  - Se B comprar, remove AvisoB, fim.
  - Se B não comprar, remove AvisoB, A pode comprar

Computadores & Leite

Esta solução funciona, mas não é boa (cara chato!):

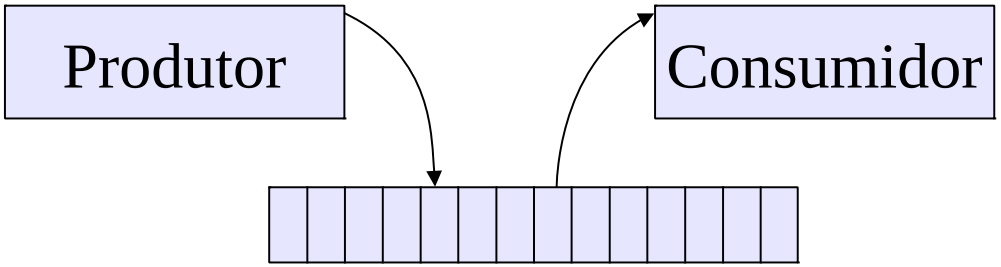
- Muito complicado. Difícil de entender e se convencer de que está correto.
- Código de A é diferente do do B. E se houver mais de dois processos?
- Enquanto A está esperando, está consumindo CPU (*busy waiting* – espera ocupada)

Computadores & Leite

Pontos importantes:

- Comportamento **muito** sutil. Difícil de programar e entender.
- Como provar que está correto?
- Quais os critérios para uma boa solução?

Produtor & Consumidor

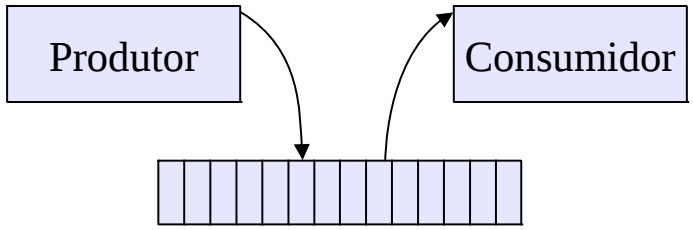


- Produtor gera itens continuamente e os coloca no *buffer*.
- Consumidor usa itens, lendo-os do *buffer.*
- *Buffer* é necessário por causa da velocidade relativa entre produtor e consumidor.
- Sincronização é necessária para acesso ao *buffer*:
  - Produtor não pode colocar mais itens em *buffer* cheio.
  - Consumidor não pode ler itens de *buffer* vazio.

Produtor & Consumidor

Solução "ideal" – usa todas as posições do *buffer*:

```
Produtor() {
    while (true) {
        while (counter == n);
        buffer[in]= item produzido;
        in= in+1 mod n;
        counter++;
    }
}
Consumidor() {
    while (true) {
        while (counter == 0);
        item consumido= buffer[out];
        out= out+1 mod n;
        counter--;
    }
}
```



- *Buffer* circular

Produtor & Consumidor

```
Produtor() {
    while (true) {
        while (counter == n);
        buffer[in]= item produzido;
        in= in+1 mod n;
        R1= counter;   INC(R1);   counter= R1;
    }
}
Consumidor() {
    while (true) {
        while (counter == 0);
        item consumido= buffer[out];
        out= out+1 mod n;
        R1= counter;   DEC(R1);   counter= R1;
    }
}
```

Produtor & Consumidor

Solução não ideal – não usa todas as posições do *buffer*:

```
Produtor() {
    while (true) {
        while (in+1 mod n == out);
        buffer[in]= item produzido;
        in= in+1 mod n;
    }
}

Consumidor() {
    while (true) {
        while (in == out);
        item consumido= buffer[out];
        out= out+1 mod n;
    }
}
```

Como escrever uma solução correta que usa *n* posições do *buffer*?

Produtor & Consumidor

Simples! Declare um *buffer* com n+1 posições:

- Uma posição de memória não vale o esforço de reescrever o programa!

Sincronização – Requisitos

Requisitos para uma primitiva de exclusão mútua:

- Deve permitir apenas um processo dentro da região crítica a cada instante;
- Se várias requisições são feitas ao mesmo tempo, deve permitir que um processo prossiga;
- Processos podem “entrar de férias” somente fora das regiões críticas.

Sincronização – Requisitos

Propriedades desejáveis de um mecanismo de exclusão mútua:

- Justiça (*fairness*): se vários processos estão esperando, dar acesso a todos, eventualmente;
- Eficiência: não utilizar quantidades substanciais de recursos quando estiver esperando. Em particular, evitar a *espera ocupada*;
- Simples: deve ser fácil de utilizar.

Sincronização – Requisitos

Propriedades dos processos utilizando os mecanismos necessários para manter coerência:

- Trancar sempre antes de utilizar dado compartilhado;
- Destrancar sempre que terminar o uso do dado compartilhado;
- Não trancar de novo se já tiver trancado o recurso;
- Não ficar muito tempo dentro das seções críticas:

```
while (!fim) {
    seção_não_crítica;
    lock();
    seção_crítica;
    unlock();
}
```

Resumo

- Cooperação usando variáveis compartilhadas.
- Necessita de **operações atômicas** por causa de *race conditions*.
- Implementações através de exclusão mútua:
  - Comportamento difícil de prever;
  - Tem que ser usado com critério;
  - Tem que ser correto, justo, eficiente.