MAC0438 – Programação Concorrente

Daniel Macêdo Batista

IME - USP, 9 de Abril de 2013

Produtores e consumidores

Buffers limitados
Problema dos filósofos famintos

- Produtores enviam mensagens consumidas pelos consumidores
- Há um buffer compartilhado manipulado por duas operações: `armazena` e `busca`
- Produtores rodam `armazena` e consumidores rodam `busca`

  Para garantir que mensagens não são sobrescritas e recebidas uma única vez, `armazena` e `busca` devem alternar a execução

  `armazena` deve executar primeiro

Produtores e consumidores – Solução anterior

Buffers limitados
Problema dos filósofos famintos

```
int buf, p = 0, c = 0;

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

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

Produtores e consumidores – com semáforos

Buffers limitados
Problema dos filósofos famintos

- Usar mais uma vez semáforos para sinalizar os eventos
- Esses semáforos podem ser implementados de modo a sinalizar:

  quando processos alcançarem pontos críticos de execução (Início e finalização das operações `armazena` e `recebe`); ou

  mudanças de variáveis compartilhadas (buffer cheio e buffer vazio)
- A solução apresentada considera o estado do buffer (melhor do que as entradas nas operações quando há múltiplos produtores e consumidores)

Produtores e consumidores – com semáforos

Buffers limitados
Problema dos filósofos famintos

- `empty` e `full` são dois semáforos

  Inicialmente o buffer está vazio, então `empty = 1`
  (Significa que o evento "esvazie o buffer" aconteceu)

  `full` começa valendo `0`

Produtores e consumidores – com semáforos

Buffers limitados
Problema dos filósofos famintos

- Se Produtor quer executar `armazena`, espera pelo buffer ficar vazio
- Após o Produtor chamar `armazena`, o buffer fica cheio
- Se Consumidor quer executar `recebe`, ele espera pelo buffer ficar cheio
- Após o Consumidor chamar `recebe`, o buffer fica vazio
- **Lembrando: com semáforos, um processo espera por um evento rodando `P` e sinaliza um evento rodando `V`**

Produtores e consumidores – com semáforos

Buffers limitados
Problema dos filósofos famintos

```
typeT buf;
sem empty = 1, full = 0;

process Producer [i = 1 to M] {
   while (true) {
      ...
      /* produz dados e armazena no buffer */
      P(empty);
      buf = data;
      V(full);
   }
}
```

Produtores e consumidores – com semáforos

Buffers limitados
Problema dos filósofos famintos

```
process Consumer [j = 1 to N] {
   while (true) {
      /* le o buffer e consome o resultado */
      P(full);
      result = buf;
      V(empty);
      ...
   }
}
```

Produtores e consumidores – com semáforos

Buffers limitados
Problema dos filósofos famintos

- `empty` e `full` são semáforos binários divididos porque no máximo um deles é `1` a cada instante
- É como se fossem um único semáforo dividido em dois semáforos binários

Buffers limitados

Desempenho do problema dos produtores/consumidores

- Acesso para um buffer simples
- Dados produzidos e consumidos na mesma taxa: **OK**
- Dados produzidos em rajada

  Produtor realiza computação e gera uma quantidade grande de dados: **Ruim**

  Como melhorar o algoritmo anterior?

Desempenho do problema dos produtores/consumidores

Buffers limitados
Problema dos filósofos famintos

- Buffer simples $\rightarrow$ Buffer com várias posições
- Semáforos binários $\rightarrow$ Semáforos genéricos

  Os semáforos passarão a contar os recursos utilizados
- **Inicialmente veremos a solução com apenas 1 produtor e 1 consumidor**

Estruturas de dados

Buffers limitados
Problema dos filósofos famintos

- Fila de mensagens ainda não consumidas

  Vetor com dois índices

  `buf[n]`, $n > 1$

  `front`: primeira mensagem da fila

  `rear`: primeira posição vazia

Algoritmo

- Sempre que o produtor produzir uma mensagem, armazena na posição `rear` e avança o `rear` (trata como um vetor circular)

  ```
  buf[rear]=data; rear=(rear+1)%n
  ```

- Sempre que o consumidor consumir uma mensagem, lê o que está na posição `front` e avança o `front` (também circular)

  ```
  result=buf[front]; front=(front+1)%n
  ```

Concorrência no algoritmo

Buffers limitados
Problema dos filósofos famintos

- No caso anterior (buffer simples), as execuções do consumo e da produção deviam se alternar
- Agora (buffers múltiplos), o consumo pode ser executado sempre que houver mensagem e a produção sempre que houver espaço vazio
- **Produções e consumos podem ser executados de forma concorrente**

  Mas a utilização de `P` e `V` é a mesma. A diferença está na inicialização do semáforo `empty` que agora recebe `n` e não `1`

Algoritmo para 1 produtor e 1 consumidor

Buffers limitados
Problema dos filósofos famintos

```
typeT buf;
int front = 0, rear = 0;
sem empty = n, full = 0;

process Producer {
   while (true) {
      ...
      /* produz dados e armazena no buffer */
      P(empty);
      buf[rear] = data; rear = (rear + 1) % n;
      V(full);
   }
}
```

Algoritmo para 1 produtor e 1 consumidor

Buffers limitados
Problema dos filósofos famintos

```
process Consumer {
   while (true) {
      /* le o buffer e consome o resultado */
      P(full);
      result = buf[front]; front = (front + 1) % n;
      V(empty);
      ...
   }
}
```

Algoritmo para 1 produtor e 1 consumidor

Buffers limitados
Problema dos filósofos famintos

- Uma nova utilidade para os semáforos: contagem de recursos
- Útil quando processos competem por acesso a recursos múltiplos

Algoritmo para M produtores e N consumidores

Buffers limitados
Problema dos filósofos famintos

□ Simples – Apenas criar M produtores

```
typeT buf;
int front = 0, rear = 0;
sem empty = n, full = 0;

process Producer [i=1 to M] {
   while (true) {
      ...
      /* produz dados e armazena no buffer */
      P(empty);
      buf[rear] = data; rear = (rear + 1) % n;
      V(full);
   }
}
```

Algoritmo para M produtores e N consumidores

Buffers limitados
Problema dos filósofos famintos

- **Simples – Apenas criar N consumidores**

```
process Consumer [j = 1 to N] {
  while (true) {
     /* le o buffer e consome o resultado */
     P(full);
     result = buf[front]; front = (front + 1) % n;
     V(empty);
     ...
  }
}
```

Algoritmo para M produtores e N consumidores

Buffers limitados
Problema dos filósofos famintos

- Na solução anterior, se há pelo menos 2 posições vazias no buffer

  Dois produtores armazenariam mensagens diferentes na mesma posição!

- Similar para os consumidores: leriam a mesma mensagem

- As execuções de produção e consumo são seções críticas!

  Sabemos resolver com semáforos :)

  Semáforos binários `mutex` para cada seção crítica

Algoritmo para M produtores e N consumidores

Buffers limitados
Problema dos filósofos famintos

```
typeT buf;
int front = 0, rear = 0;
sem empty = n, full = 0;
sem mutexD = 1, mutexF = 1;

process Producer [i=1 to M] {
   while (true) {
      ...
      /* produz dados e armazena no buffer */
      P(empty);
      P(mutexD);
      buf[rear] = data; rear = (rear + 1) % n;
      V(mutextD);
      V(full);
   }
}
```

Algoritmo para M produtores e N consumidores

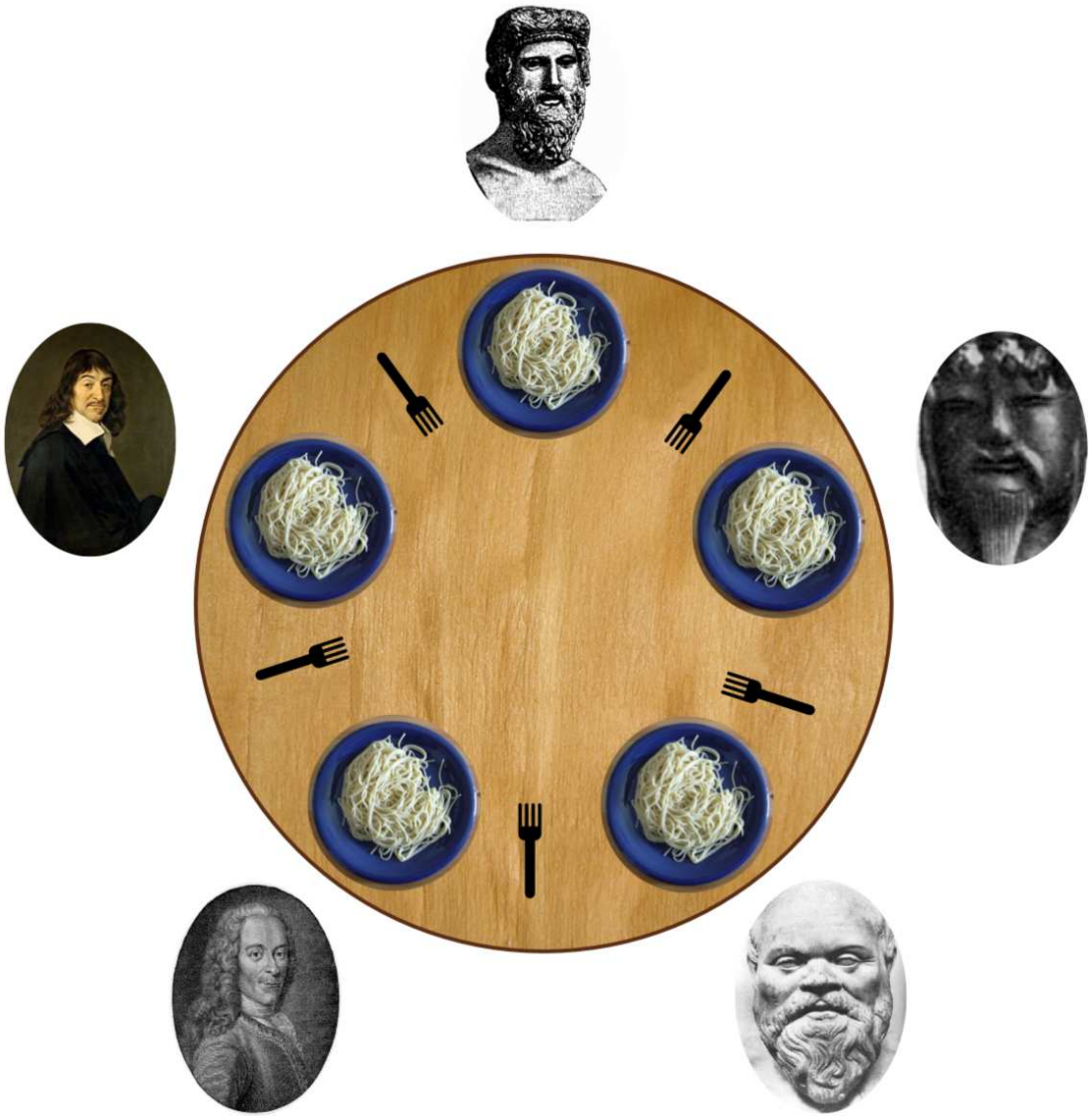
Buffers limitados
Problema dos filósofos famintos

```
process Consumer [j = 1 to N] {
  while (true) {
     /* le o buffer e consome o resultado */
     P(full);
     P(mutexF);
     result = buf[front]; front = (front + 1) % n;
     V(mutexF);
     V(empty);
     ...
  }
}
```

Problema dos filósofos famintos

Descrição

Buffers limitados
Problema dos filósofos famintos



Descrição

- 5 filósofos
- Comem e pensam
- Cada filósofo tem que usar 2 garfos para comer
- Há apenas 5 garfos
- Cada filósofo só pode usar os garfos imediatamente na sua esquerda e na sua direita

Objetivo

- O objetivo é implementar um programa que simule o comportamento dos filósofos
- **O programa deve evitar a situação em que todos os filósofos estejam com fome mas não consigam adquirir ambos os garfos – Por exemplo, cada um segura um garfo e se recusa a liberá-lo**
- Implementar exclusão mútua entre processos que competem por conjuntos de variáveis compartilhadas que se sobrepõem
- Útil quando um processo requer acesso simultâneo a mais de um recurso

Mais detalhes do problema

- Algumas características do problema:

  Dois filósofos vizinhos não podem comer ao mesmo tempo

  No máximo 2 filósofos estarão comendo ao mesmo tempo

- Algumas considerações do problema:

  Os períodos pensando e comendo podem variar (aleatório)

Algoritmo dos filósofos

Buffers limitados
Problema dos filósofos famintos

```
process Philosopher [i = 0 to 4] {
   while (true) {
      pensa;
      pega os garfos;
      come;
      libera os garfos;
   }
}
```

Algoritmo dos filósofos

Buffers limitados
Problema dos filósofos famintos

- Cada garfo age como uma trava de seção crítica

  Só pode estar com um filósofo por vez

  Os garfos serão um vetor de semáforos inicializados com 1 (inicialmente ninguém segura nenhum garfo)

- Com semáforos

  Pegar um garfo = executar `P`

  Soltar um garfo = executar `V`

Primeira solução

- Tentar ações idênticas
- Por exemplo, cada filósofo pega (tenta) primeiro o garfo da esquerda e depois o da direita
- **Resolve o problema?**

Segunda solução

- Evitar o deadlock que a solução anterior permitia
- Evitar a espera circular (o primeiro espera o segundo, que espera o terceiro, ... que espera o primeiro)
- **Ideias?**

Segunda solução

- Quebrar a espera circular
- Fazer algum filósofo tentar pegar o garfo da direita primeiro

```
sem fork[5] = {1, 1, 1, 1, 1};

process Philosopher [i = 0 to 3] {
   while (true) {
      P(fork[i]);P(fork[i+1]);
      come;
      V(fork[i]);V(fork[i+1]);
      pensa;
   }
}
```

Segunda solução


```
process Philosopher [4] {
  while (true) {
     P(fork[0]);P(fork[4]);
     come;
     V(fork[0]);V(fork[4]);
     pensa;
  }
}
```