
MAC0422 - Sistemas Operacionais

Daniel Macêdo Batista

IME - USP, 1 de Outubro de 2020

Problemas clássicos – Buffers limitados

Problemas clásicos – Buffers limitados

Desempenho do problema dos produtores/consumidores

Problemas clássicos
– Buffers limitados

- 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

Problemas clássicos
– Buffers limitados

- Buffer simples → Buffer com várias posições
- Semáforos binários → Semáforos genéricos
 - Os semáforos passarão a contar os recursos utilizados
- Inicialmente veremos a solução com apenas 1 produtor e 1 consumidor**

Estruturas de dados

Problemas clássicos
– Buffers limitados

- Fila de mensagens ainda não consumidas
Vetor com dois índices
 $\text{buf}[n], n > 1$
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

- 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

Problemas clássicos
– Buffers limitados

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

Thread 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

Problemas clássicos
– Buffers limitados

```
Thread 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

- 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

□ **Simplex – Apenas criar M produtores**

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

□ **Simplex – Apenas criar N consumidores**

```
Thread 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

- 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

```
typeT buf;  
int front = 0, rear = 0;  
sem empty = n, full = 0;  
sem mutexD = 1, mutexF = 1;  
  
Thread 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(mutexD);  
        V(full);  
    }  
}
```

Algoritmo para M produtores e N consumidores

Problemas clássicos
– Buffers limitados

```
Thread 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);  
        ...  
    }  
}
```