BCC722

Programação de Sistemas em Tempo Real

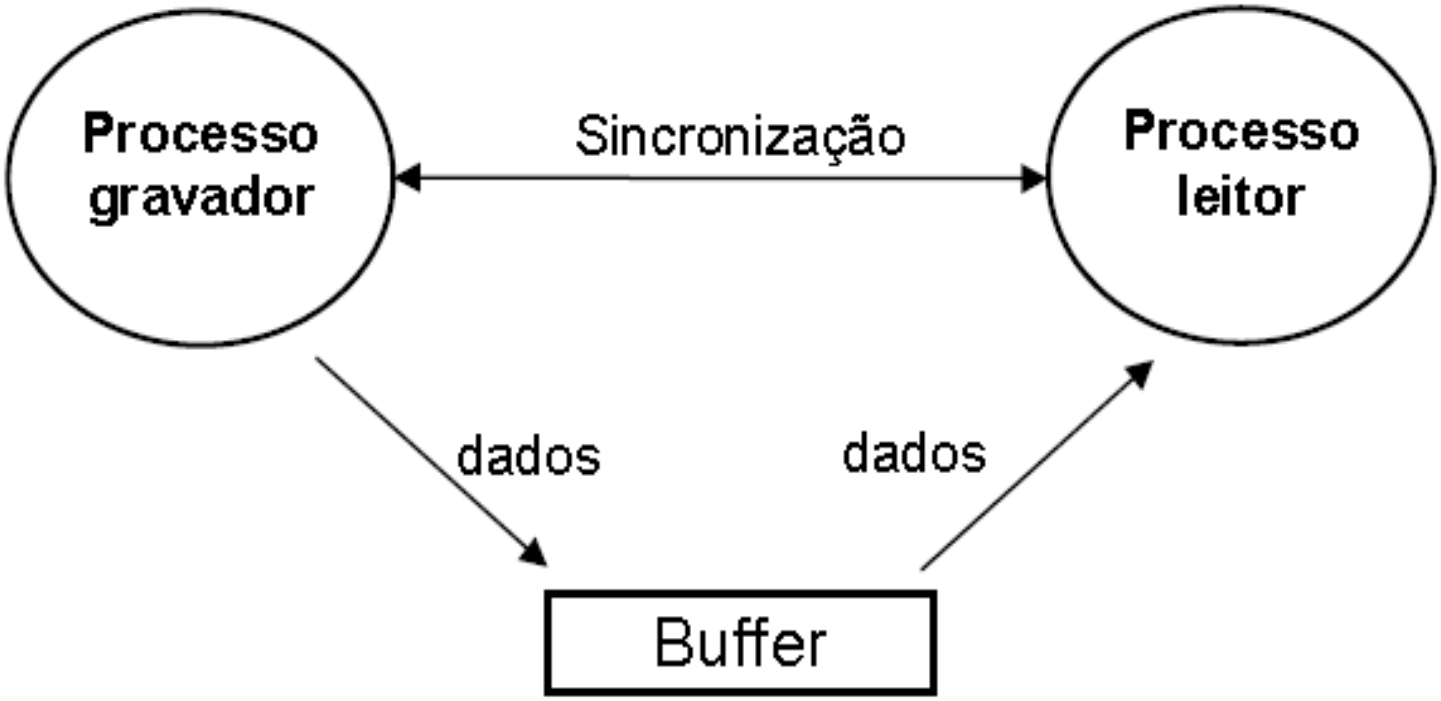
Concorrência entre Processos

Prof. Charles Garrocho

Antecedentes

- Processo Cooperativo é aquele que pode afetar outros processos em execução, ou ser por eles afetados.
- Acesso concorrente a dados compartilhados pode resultar em inconsistências
- Manter dados consistentes exige mecanismos para garantir a execução cooperativa de processos

O problema do Buffer limitado (mem. compartilhada)



Buffer limitado: produtor

```
item nextProduced;
while (1) {
  while (counter == BUFFER_SIZE)
     /* buffer cheio */
     /* não faz coisa alguma */ ;

  buffer[counter] = nextProduced;
  counter++;
}
```

Buffer limitado: consumidor

```
item nextConsumed;
while (1) {
   while (counter==0)
   /* buffer vazio */
   /* não faz coisa alguma */ ;

   nextConsumed = buffer[counter];
   counter--;
}
```

Buffer limitado (mem. compartilhada)

- Os comandos

```
counter++; /* produtor */
counter--; /* consumidor */
```

  precisam ser executados de forma atômica.
  - isto é: não podem ser interrompidos no meio de sua execução.

Buffer limitado (mem. compartilhada)

- Código para `count++` em assembly:

```
a)MOV R1,$counter
b)INC R1
c)MOV $counter,R1
```

- Código para `count--` em assembly:

```
x)MOV R2, $counter
y)DEC R2
z)MOV $counter, R2
```

Buffer limitado (mem. compartilhada)

- Seja *counter* igual a 5 inicialmente e considere-se a seguinte sequência de execução:

produtor:	`MOV R1,$counter`	(R1 = 5)
produtor:	`INC R1`	(R1 = 6)
consumidor:	`MOV R2,$counter`	(R2 = 5)
consumidor:	`DEC R2`	(R2 = 4)
produtor:	`MOV $counter,R1`	(counter = 6)
consumidor:	`MOV $counter,R2`	(counter = 4)

- O valor de *counter* pode terminar como 4 ou 6, mas o correto seria 5!

Condição de corrida

- Situação onde vários processos acessam e manipulam os mesmos dados de forma concorrente
- Sincronização entre processos pode ser usada para evitar tais corridas

O problema de seção crítica

- N processos competem para usar alguma estrutura de dados compartilhada
- Cada processo tem um segmento de código comum onde a estrutura é acessada
- Problema: garantir que quando um processo está executando aquele segmento de código, nenhum outro processo pode fazer o mesmo

O problema de seção crítica

- Requisitos da solução procurada:
  - Exclusão mútua: se Pi está na seção crítica, nenhum outro processo pode entrar nela.
  - Progresso: os processos participam na decisão sobre qual é o próximo processo a entrar na seção crítica.
  - Espera limitada: se um processo deseja entrar na seção crítica, há um limite no num. de outros processos que podem entrar nela antes dele.

Estrutura geral de um processo típico Pi

- Só dois processos, Pi e Pj

```
do {
   enter section
   critical section
   leave section
   remainder section
} while (1);
```

- Processos podem compartilhar algumas variáveis para conseguir o controle desejado

Algoritmo 1

- Variável compartilhada: `int vez = i;`
- Processo i:

```
do {
  while (vez != i);
  critical section
  vez = j;
  remainder section
} while (1);
```

- Satisfaz exclusão mútua, mas não progresso.

Algoritmo 1

- Variável compartilhada: `int vez = i;`
- Processo j:

```
do {
  while (vez != j);
  critical section
  vez = i;
  remainder section
} while (1);
```

- Satisfaz exclusão mútua, mas não progresso.

Algoritmo 2

- Variável compartilhada: `int flag[2]={0,0};`
- Processo i:

```
do {
    flag[i]=true;
    while (flag[j]);
    critical section
    flag[i]=false;
    remainder section
} while (1);
```

- Satisfaz exclusão mútua, mas não progresso.

Algoritmo 3: combina vez e flag

- Processo i:

```
do {
        flag[i]=true; vez=j;
        while (flag[j] && vez==j);
        critical section
        flag[i]=false;
        remainder section
} while (1);
```

- Satisfaz os três requisitos

Hardware de sincronização

- Para simplificar algoritmos, hardware pode prover operação atômica de leitura+escrita
- Exemplo: testa e modifica conteúdo da mem.

```
boolean TestAndSet(boolean* target)
{
   boolean old_value = *target;
   *target = true;
   return old_value;
}
```

Exclusão mútua com TestAndSet

- Variável compartilhada: boolean `lock=false;`
- Processo i:

```
do {
   while (TestAndSet(&lock));
   critical section
   lock = false;
remainder section
} while (1);
```

Hardware de sincronização

- Outro exemplo: troca conteúdo de duas posições

```
void Swap(boolean* a, boolean* b)
{
    boolean tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Exclusão mútua com Swap

- Variável compartilhada: `boolean lock=false;`
- Processo i (possui variável local `key`):

```
do {
    key = true;
    while (key)
      Swap(&lock,&key);
   critical section
    lock = false;
remainder section
 } while (1);
```

Exclusão mútua por hardware

- Esses algoritmos garantem progresso, mas não espera limitada
- `Solução: cada processo registra sua intenção de entrar na região crítica separadamente`
  - `boolean waiting[n]; // = false`

Ao sair da região crítica, processo “passa a vez” diretamente ao próximo, se ele existir

Exclusão mútua por hardware

```
do {
   waiting[i] = true;
   while (waiting[i] && TestAndSet(&lock)); // Exclusao Mutua
   critical section
   j = next(i);
   while ((j!=i) && !waiting[j]) // procura o proximo
        j = next(j);  // Espera limitada
   if (j==i) { lock = false; } // ninguém esperando
   else { waiting[j] = false; } // passa a vez diretamente
   waiting[i] = false;  // Progresso
   remainder section
} while (1);
```

Semáforos

- Variável inteira acessível apenas através de duas operações indivisíveis (atômicas):

```
wait(int* s) {
    while (*s<=0);
    (*s)--;
}
```

```
signal(int* s) {
    (*s)++;
}
```

Problema do buffer limitado (produtor/consumidor)

- Estruturas de dados:

```
sem mutex = 1;
sem nbuffempty = TAM_BUF;
sem nbuffull = 0;
```

```
produtor:
do {
  // produz um item
  wait(nbuffempty);
  wait(mutex);
  // item → buffer
  signal(mutex);
  signal(nbuffull);
} while (1);
```

```
consumidor:
do {
  wait(nbuffull);
  wait(mutex);
  // item ← buffer
  signal(mutex);
  signal(nbuffempty);
  // processa item
} while (1);
```

Regiões críticas

- Abstração de alto nível para sincronização focada nos dados
- Uma variável compartilhada é declarada como tal
  - v: shared T
- Essa variável só pode ser acessada em regiões
  - region v when B do S // B: expressão booleana

Regiões críticas: implementação do *buffer* limitado

- Estruturas de dados:

```
shared struct {
  item pool[n]
  int count, in, out;
} buf;
```

produtor:

```
// produz item
region buf
  when (count < n) {
    pool[in] = nitem;
    in = (in+1) % n;
    count++;
}
```

consumidor:

```
region buf
  when (count > 0) {
    nitem = pool[out];
    out = (out+1) % n;
    count--;
}
// consome item
```

Transações atômicas

- Conceito originário da área de bancos de dados
- Define um conjunto de operações que deve ser executado atomicamente:
  - Ou todas operações executam, ou nenhuma
  - P.ex.: saque de uma conta bancária
- Operação normalmente baseada em logs
  - Registra-se a operação que vai ser realizada
  - Executa-se (ou não) a operação
  - O log pode ser consultado no caso de falhas