MC504 - Sistemas Operaconais

Sincronização: da espera ocupada a semáforos

Islene Calciolari Garcia
Instituto de Computação - Unicamp
Primeiro Semestre de 2021

Sumário

Revisão: Multithread e recursos compartilhados

- Multithread: forma mais natural para implementar alguns problemas e possibilidade de paralelismo
- Exemplos de recursos a serem compartilhados: estruturas de dados, arquivos, dispositivos etc.
- Estudo de caso com variável inteira:

```
volatile int s; /* Variável compartilhada */

/* As threads da aplicação podem ler ou
   escrever na variável s */

s = thr_id;
```

Revisão: O que é espera ocupada?

```
ready = 0;
while (!ready) ;
```

- loop potencialmente infinito
- `ready` deve ser alterada por outra thread ou dispositivo
- consome muita CPU enquanto espera :-(

```
$ ./espera_ocupada &
$ top
$ killall ./espera_ocupada
```

- NÃO UTILIZAR ESTA ESTRATÉGIA NO PROJETO 1

Revisão: Algoritmos de exclusão mútua

- Acesso controlado a recursos compartilhados
- Estudo de caso:

```
volatile int s; /* Variável compartilhada */
  while (1) {
      /* Região não crítica */
      /* Protocolo de entrada */
      /* Região crítica */
        s = thr_id;
        printf ("Thr %d: %d", thr_id, s);
      /* Protocolo de saída */
  }
```

Revisão: Exclusão Mútua

- Os algoritmos devem garantir:
  - exclusão mútua
  - ausência de *deadlock*
  - ausência de *starvation*
  - progresso (uma thread que não esteja interessada na região crítica não pode impedir outra thread de entrar na região crítica)

Revisão: Alternância (passagem de *token*)

Progresso limitado :-(

```
int s = 0;
int vez = 1; /* Primeiro a thread 1 */
```

Thread 0

```
while (true)
  while (vez != 0);
  s = 0;
  print ("Thr 0:" , s);
  vez = 1;
```

Thread 1

```
while (true)
  while (vez != 1);
  s = 1;
  print ("Thr 1:" , s);
  vez = 0;
```

- Veja o código: `../aula03/alternancia.c`

Revisão: Vetor de Interesse

Possibilidade de deadlock :-(

```
int s = 0;
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  while (interesse[1]);
  s = 0;
  print("Thr 0:" , s);
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  while (interesse[0]);
  s = 1;
  print("Thr 1:" , s);
  interesse[1] = false;
```

- Veja o código: `../aula03/interesse.c`

Revisão: Interesse e Alternância

Não garante exclusão mútua :-(

```
int s = 0, vez = 0;
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  if (interesse[1])
    while (vez != 0);
  s = 0;
  print("Thr 0:", s);
  vez = 1;
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  if (interesse[0])
    while (vez != 1);
  s = 1;
  print("Thr 1:", s);
  vez = 0;
  interesse[1] = false;
```

- Veja o código: `../aula03/interesse_vez.c`

Vetor de Interesse e Alternância

Não garante exclusão mútua :-(

```
int s = 0, vez = 0;
int interesse[2] = {false, false};
```

Thread 0

Thread 1

```
while (true)
  interesse[1] = true;
  if (interesse[0])
    while (vez != 1);
  s = 1;
```

Revisão: Vetor de Interesse e Alternância

Não garante exclusão mútua :-(

```
int s = 0, vez = 0;
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  if (interesse[1])
    while (vez != 0);
  s = 0;
```

Thread 1

```
while (true)
  interesse[1] = true;
  if (interesse[0])
    while (vez != 1);
  s = 1;
```

Revisão: Algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
  interesse[0] = true;
  while (interesse[1])
    if (vez != 0)
      interesse[0] = false;
      while (vez !=0);
      interesse[0] = true;
  s = 0;
  print ("Thr 0:" , s);
  vez = 1;
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  while(interesse[0])
    if (vez != 1)
      interesse[1] = false;
      while(vez != 1);
      interesse[1] = true;
  s = 1;
  print ("Thr 1:" , s);
  vez = 0;
  interesse[1] = false;
```

Dekker para N threads?

Mesmo problema da abordagem da alternância

```
int vez = 0, interesse = {false, ..., false}
while (true) {  /* Código da Thread_i */
   interesse[i] = true;
   while (existe j!=i tal que (interesse[j]))
     if (vez != i)
       interesse[i] = false;
       while (vez != i);
       interesse[i] = true;
   s = i;
   print ("Thr ", i, ": ", s);
   vez = (vez + 1) % N;
   interesse[i] = false;
```

Dekker para N threads?

E se passássemos a vez para a próxima interessada?

```
int vez = 0, interesse = {false, ..., false}
while (true) {  /* Código da Thread_i */
   interesse[i] = true;
   while (existe j!=i tal que (interesse[j]))
     if (vez != i)
       interesse[i] = false;
       while (vez != i);
       interesse[i] = true;
   s = i;
   print ("Thr ", i, ": ", s);
   vez = próxima_interessada
   interesse[i] = false;
```

Algoritmo de Dijkstra (1965)

```
int vez = -1, interesse = {false, ..., false}
while (true) {  /* Código da Thread_i */
   interesse[i] = true;
   while (existe j!=i tal que (interesse[j]))
     if (vez != i)
       interesse[i] = false;
       while (vez != -1);
       vez = i;
       interesse[i] = true;
   s = i;
   print ("Thr ", i, ": ", s);
   vez = -1
   interesse[i] = false;
```

Algoritmo de Dijkstra

Garante exclusão mútua?

- Uma thread só entra na região crítica após percorrer o vetor e verificar que nenhuma outra está interessada

Garante ausência de deadlock?

- Entre as interessadas, pelo menos a última a alterar a variável vez consegue entrar na região crítica
- Não há cenário de bloqueio indefinido

Garante ausência de starvation?

- Não. Uma thread pode nunca conseguir ser a última a alterar vez.
- Veja o código `dijkstra.c`

Algoritmo de Dijkstra

Como ilustrar o problema de starvation?

- Trecho de código válido:

```
if (thr_id == 3)
   sleep(1);
```

- Trechos de código inválido:

```
if (thr_id == 3)
   sleep(1000000); /* dorme para sempre e
                      morre de fome... */
if (thr_id != 3)
  vez = i;    /* Nunca passa a vez para thread 3 */
```

Algoritmo de Dijkstra

Solução de Andre Macedo, Marcelo O. de Moraes e Thaís A. B. Fernandes

```
interesse[thr_id] = 0;
while (vez != -1);
/* Garante que a thread 1 sempre é a primeira */
if( thr_id == 1 )
    sleep( 1 );
else
    sleep( 2 );
vez = thr_id;
/* Sincroniza novamente todas as threads */
if( thr_id == 1 )
   sleep( 2 );
else
   sleep( 1 );
interesse[thr_id] = 1;
```

Algoritmo do Desempate (1981)

```
int s = 0, ultimo = 0, interesse[2] = {false, false};
```

Thread 0

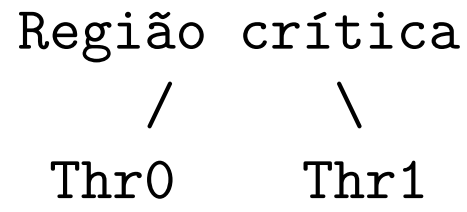
```
while (true)
  interesse[0] = true;
  ultimo = 0;
  while (ultimo == 0 &&
         interesse[1]);
  s = 0;
  print ("Thr 0:" , s);
  interesse[0] = false;
```

Thread 1

```
while (true)
  interesse[1] = true;
  ultimo = 1;
  while (ultimo == 1 &&
         interesse[0]);
  s = 1;
  print ("Thr 1:" , s);
  interesse[1] = false;
```

Algoritmo do Desempate

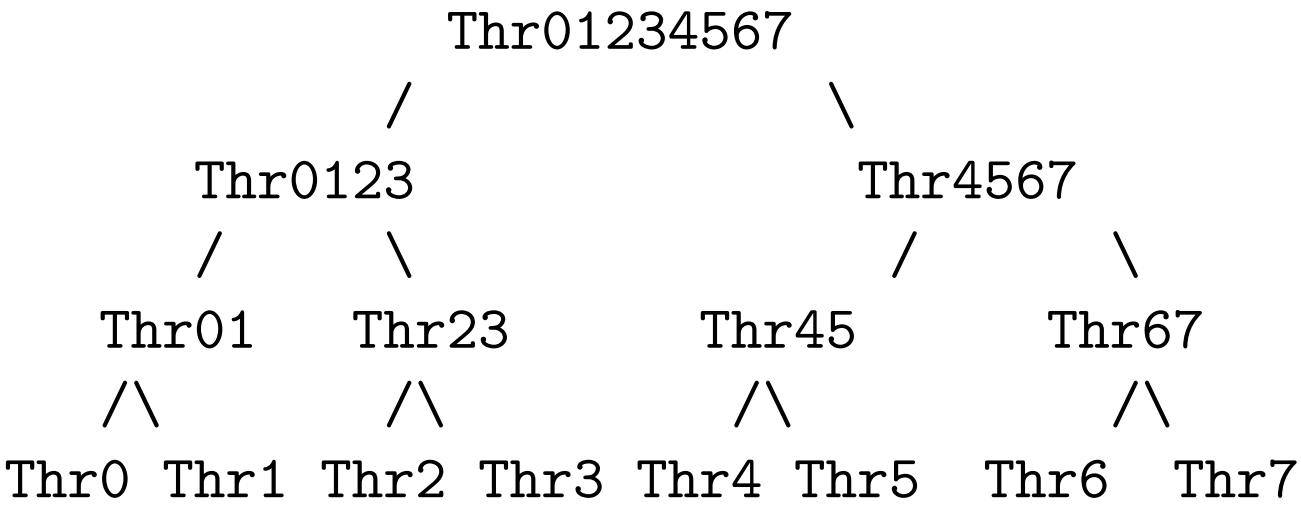
Características



- Funciona para 2 threads
- Variável ultimo é acessada pelas 2 threads
- Variável interesse[i] é acessada
  - para escrita pela thread i
  - para leitura pela thread adversária

Exclusão mútua entre N threads

Abordagem do campeonato



- As threads podem concorrer duas a duas
- Garante ausência de starvation?
- **Como fica a justiça nesta abordagem?**

Campeonato e justiça

4 threads, todas interessadas

```
Thr 0
Thr 2
Thr 1
Thr 3
Thr 0
Thr 2
Thr 1
Thr 3
Thr 0
Thr 2
Thr 1
Thr 3
...
```

Campeonato e justiça

4 threads, apenas 3 interessadas

```
Thr 0
Thr 2
Thr 1
Thr 2
Thr 0
Thr 2
Thr 1
Thr 2
Thr 0
Thr 2
Thr 1
Thr 2
...
```

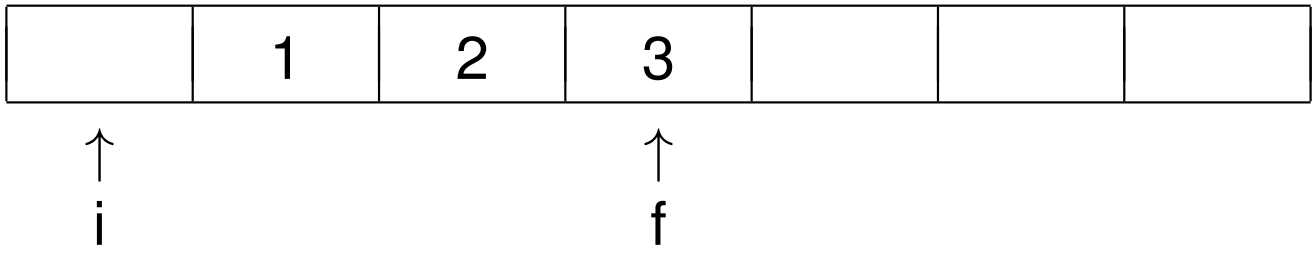
Problema do Produtor-Consumidor

- Dois processos compartilham um *buffer* de tamanho fixo
- O produtor insere informação no *buffer*
- O consumidor remove informação do *buffer*

Exemplo de saída consistente

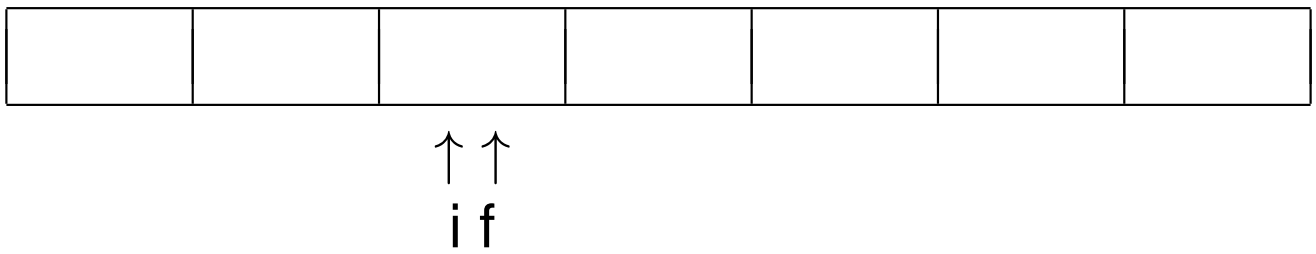
```
Produtor:   item_0
Consumidor: item_0
Produtor:   item_1
Produtor:   item_2
Consumidor: item_1
Produtor:   item_3
Consumidor: item_2
Consumidor: item_3
Produtor:   item_4
Consumidor: item_4
Produtor:   item_5
Consumidor: item_5
```

Controle do buffer



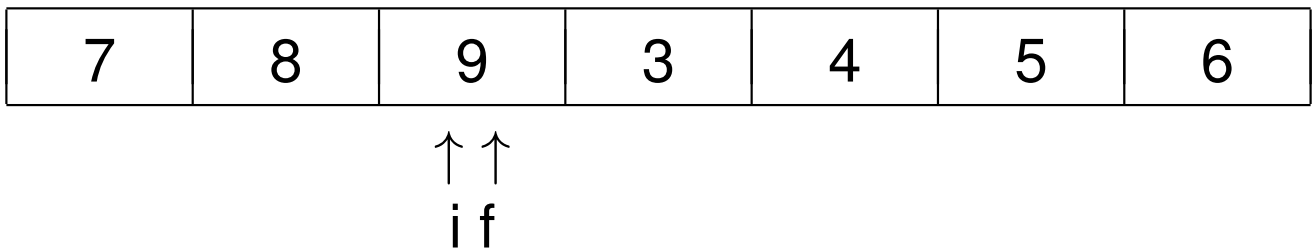
- **i:** posição anterior ao próximo elemento a ser consumido
- **f:** último elemento produzido
- **c:** indica o número de elementos presentes
- **N:** indica o número máximo de elementos

Buffer vazio



- **i == f**
- **c == 0**

Buffer cheio



- **i == f**
- **c == N**

Implementação sem sincronização

```
int buffer[N];
 int c = 0;
 int i = 0, f = 0;
```

Produtor

```
 while (true)
   f = (f+1)%N;
   buffer[f]= produz();
   c++;
```

Consumidor

```
 while (true)
   i = (i+1)%N;
   consome(buffer [i]);
   c--;
```

Veja código: `prod-cons-sem-sinc.c`

Exemplo de saída com *Bug*

Consumidor lê dados antes da escrita

```
Produtor:   item_0
Consumidor: item_0
Consumidor: item_1

Erro: item foi consumido antes de ser produzido

Produtor:   item_1
Consumidor: item_2

Erro: item foi consumido antes de ser produzido.
```

Exemplo de saída com *Bug*

Produtor sobrescreve posição que *ainda* não foi consumida

```
Produtor:   item_0
Produtor:   item_1
Produtor:   item_2
Produtor:   item_3
Produtor:   item_4.
Produtor:   item_5  (Suponha sobrescrita da posição 0)
Consumidor: item_5

Erro: item foi produzido em posição ocupada
```

Algoritmo com espera ocupada

```
int buffer[N];
int c = 0, i = 0, f = 0;
```

Produtor

```
while (true)
  while (c == N);
  f = (f+1)%N;
  buffer[f]= produz();
  c++;
```

Consumidor

```
while (true)
  while (c == 0);
  i = (i+1)%N;
  consome(buffer[i]);
  c--;
```

Veja código: `prod-cons-espera-ocupada.c`

Possibilidade de inconsistência

Produtor	**Consumidor**
`c++;`	`c--;`
`mov rp,c`	`mov rc,c`
`inc rp`	`dec rc`
`mov c,rp`	`mov c,rc`

- Decremento/incremento não são atômicos
- Veja o código: `prod-cons-race.c`

Possibilidade de inconsistência

```
c = 5
Produtor: c++
Consumidor: c--
Valor final esperado para c: 5
```

Produtor	Consumidor	
mov rp, c		rp = 5
	mov rc,c	rc = 5
inc rp		rp = 6
	dec rc	rc = 4
mov c, rp		c = 6
	mov c, rc	c = 4

Algoritmo com espera ocupada e operações atômicas

```
int buffer[N];
int c = 0, i = 0, f = 0;
```

Produtor

```
while (true)
  while (c == N);
  f = (f+1)%N;
  buffer[f]= produz();
  atomic_inc(c);
```

Consumidor

```
while (true)
  while (c == 0);
  i = (i+1)%N;
  consome(buffer[i]);
  atomic_dec(c);
```

Veja código: `prod-cons-espera-ocupada.c`

Dormir ao invés de testar continuamente

```
int buffer[N];  int c = 0, i = 0, f = 0;
```

Produtor

```
while (true)
  if (c == N) sleep();
  f = (f + 1)%N;
  buffer[f]= produz();
  atomic_inc(c);
  if (c == 1)
    wakeup_consumidor();
```

Consumidor

```
while (true)
  if (c == 0) sleep();
  i = (i+1)%N;
  consome(buffer [i]);
  atomic_dec(c);
  if (c == N - 1)
    wakeup_produtor();
```

Possibilidade de Lost Wake-Up

```
int buffer[N];  int c = 0, i = 0, f = 0;
```

Produtor

```
while (true)
  if (c == N) sleep();
  f = (f + 1)%N;
  buffer[f]= produz();
  atomic_inc(c);
  if (c == 1)
    wakeup_consumidor();
```

Consumidor

```
while (true)
  if (c == 0) sleep();
  i = (i+1)%N;
  consome(buffer [i]);
  atomic_dec(c);
  if (c == N - 1)
    wakeup_produtor();
```

- Produtor envia sinal antes de o consumidor ir dormir
- Consumidor envia sinal antes de o produtor ir dormir

Semáforos

- Semáforos são *contadores especiais* para recursos compartilhados.
- Proposto por Dijkstra (1965)
- Operações básicas (atômicas):
  - decremento (down, wait ou P)
    bloqueia se o contador for nulo
  - incremento (up, signal (post) ou V)
    nunca bloqueia

Semáforos

Comportamento básico

- sem_init(s, 5)
- wait(s)

```
if (s == 0)
   bloqueia_processo();
else s--;
```

- signal(s)

```
if (s == 0 && existe processo bloqueado)
  acorda_processo();
else s++;
```

- **Importante:** os desenvolvedores deverão garantir uma implementação correta utilizando instruções atômicas ou primitivas de sincronização de mais baixo nível.

Produtor-Consumidor com Semáforos

```
semaforo cheio = 0;
semaforo vazio = N;
```

Produtor:

```
while (true)
    wait(vazio);
    f = (f+1)%N;
    buffer[f] = produz();
    signal(cheio);
```

Consumidor:

```
while (true)
    wait(cheio);
    i = (i+1)%N;
    consome(buffer[i]);
    signal(vazio);
```

Veja código: `prod-cons-semaforo.c`

Próximas aulas

- Problemas clássicos de sincronização
- Implementação de semáforos da `glibc`