

MC504/MC514 - Sistemas Operacionais

Problema dos Produtores e Consumidores Primitivas de Sincronização

Islene Calciolari Garcia

Segundo Semestre de 2013

Sumário

- 1 Objetivos
- 2 Introdução
- 3 Espera ocupada
- 4 Futexes
- 5 Semáforos
- 6 Mutexes

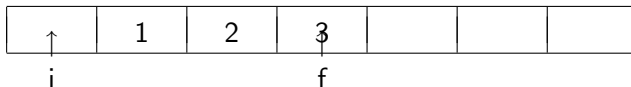
Objetivos

- Problema do produtor-consumidor
- Primitivas de sincronização
 - Semáforos
 - Locks e variáveis de condição

Problema do Produtor-Consumidor

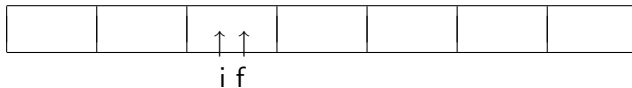
- Dois processos compartilham um *buffer* de tamanho fixo
- O produtor insere informação no *buffer*
- O consumidor remove informação do *buffer*

Controle do buffer



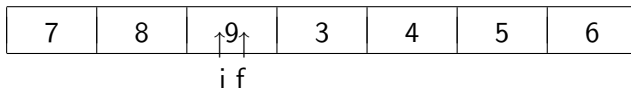
- **i:** aponta para a posição anterior ao primeiro elemento
- **f:** aponta para o último elemento
- **c:** indica o número de elementos presentes
- **N:** indica o número máximo de elementos

Buffer vazio



- $i == f$
- $c == 0$

Buffer cheio



- $i == f$
- $c == N$

Comportamento básico

```
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-basico.c

Problemas

- 1 produtor insere em posição que *ainda* não foi consumida
- 2 consumidor remove de posição *já* foi consumida

Algoritmo com espera ocupada

```
int buffer[N];  
int c = 0;  
int 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-basico-busy-wait.c

Condição de disputa

Produtor

```
c++;  
mov rp,c  
inc rp  
mov c,rp
```

Consumidor

```
c--;  
mov rc,c  
dec rc  
mov c,rc
```

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

Operações atômicas

- Veja info gcc - C extensions - Atomic builtins
- Veja o código prod-cons-basico-atomic-inc.c

Possibilidade de Lost Wake-Up

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

Produtor

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

Consumidor

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

Futex: Prototype

```
long sys_futex (  
    void *addr1,  
    int op,  
    int val1,  
    struct timespec *timeout,  
    void *addr2,  
    int val3);  
  
int syscall(SYS_futex, addr1, FUTEX_XXXX,  
            val1, timeout, addr2, val3);
```

FUTEX_WAIT

```
/* Retorna -1 se o futex não bloqueou e  
   0 caso contrário */  
int futex_wait(void *addr, int val1) {  
    return syscall(SYS_futex, addr, FUTEX_WAIT,  
                   val1, NULL, NULL, 0);} 
```

- Bloqueio até notificação
- Não há bloqueio se `*addr1 != val1`
- Veja o código `ex0.c`

FUTEX_WAKE

```
/* Retorna o número de threads acordadas */  
int futex_wake(void *addr, int n) {  
    return syscall(SYS_futex, addr, FUTEX_WAKE,  
                   n, NULL, NULL, 0);};
```

- Quantas threads acordar?
 - 1
 - 5
 - INT_MAX (todas)
- Veja os códigos ex1.c e ex2.c

Futex e operações atômicas

- Veja o código `prod-cons-basico-futex.c`
- O algoritmo não é tão simples para vários produtores e vários consumidores

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++;
```
- Veja a implementação da glibc: `sem_wait.c` e `sem_post.c`

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-sem.c

Vários produtores e consumidores

```
semaforo cheio = 0, vazio = N;  
semaforo lock_prod = 1, lock_cons = 1;
```

Produtor:

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

Consumidor:

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

Vários produtores e consumidores

```
semaforo cheio = 0, vazio = N;  
semaforo lock_prod = 1, lock_cons = 1;
```

Produtor:

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

Consumidor:

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

Semáforos

- Exclusão mútua
- Sincronização

Mutex locks

⇒ Exclusão mútua

- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Variáveis de condição

⇒ Sincronização

- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- precisam ser utilizadas em conjunto com `mutex_locks`

Thread 0 acorda Thread 1

```
int s;                                /* Veja cond_signal.c */
```

Thread 1:

```
mutex_lock(&mutex);  
if (preciso_esperar(s))  
    cond_wait(&cond, &mutex);  
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);  
if (devo_acordar_thread_1(s))  
    cond_signal(&cond);  
mutex_unlock(&mutex);
```

Produtor-Consumidor

```
int c = 0; /* Contador de posições ocupadas */  
mutex_t lock_c; /* lock para o contador */  
  
cond_t pos_vazia; /* Para o produtor esperar */  
cond_t pos_ocupada; /* Para o consumidor esperar */
```

Produtor-Consumidor

```
int f = 0;
```

Produtor:

```
mutex_lock(&lock_c);  
if (c == N)  
    cond_wait(&pos_vazia, &lock_c);  
f = (f+1)%N;  
buffer[f] = produz();  
c++;  
if (c == 1)  
    cond_signal(&pos_ocupada);  
mutex_unlock(&lock_c);
```

Produtor-Consumidor

```
int i = 0;
```

Consumidor:

```
mutex_lock(&lock_c);  
if (c == 0)  
    cond_wait(&pos_ocupada, &lock_c);  
i = (i+1)%N;  
consome(buffer[i]);  
if (c == N-1)  
    cond_signal(&pos_vazia);  
c--;  
mutex_unlock(&lock_c);
```

Produtor-Consumidor

```
cond_t pos_vazia, pos_ocupada; mutex_t lock_v, lock_o;  
int i = 0, f = 0, nv = N, no = 0;
```

Produtor:

```
    mutex_lock(&lock_v);  
    if (nv == 0) cond_wait(&pos_vazia, &lock_v);  
    nv--;  
    mutex_unlock(&lock_v);  
    f = (f+1)%N;  
    buffer[f] = produz();  
    mutex_lock(&lock_o);  
    no++;  
    cond_signal(&pos_ocupada);  
    mutex_unlock(&lock_o);
```

Produtor-Consumidor

Consumidor:

```
mutex_lock(&lock_o);  
if (no == 0) cond_wait(&pos_ocupada, &lock_o);  
no--;  
mutex_unlock(&lock_o);  
i = (i+1)%N;  
consome(buffer[i]);  
mutex_lock(&lock_v);  
nv++;  
cond_signal(&pos_vazia);  
mutex_unlock(&lock_v);
```

Produtores-Consumidores

```
cond_t pos_vazia, pos_ocupada; mutex_t lock_v, lock_o;  
int i = 0, f = 0, nv = N, no = 0;
```

Produtor:

```
    mutex_lock(&lock_v);  
    while (nv == 0) cond_wait(&pos_vazia, &lock_v);  
    nv--;  
    mutex_unlock(&lock_v);  
    f = (f+1)%N;  
    buffer[f] = produz();  
    mutex_lock(&lock_o);  
    no++;  
    cond_signal(&pos_ocupada);  
    mutex_unlock(&lock_o);
```


Produtor-Consumidor

Consumidor:

```
mutex_lock(&lock_o);  
while (no == 0) cond_wait(&pos_ocupada, &lock_o);  
no--;  
mutex_unlock(&lock_o);  
i = (i+1)%N;  
consome(buffer[i]);  
mutex_lock(&lock_v);  
nv++;  
cond_signal(&pos_vazia);  
mutex_unlock(&lock_v);
```