



Sistemas Operacionais

Problemas clássicos de comunicação entre processos

Prof. Fernando Parente Garcia



Produtor-Consumidor

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item( );
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item( );
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```



Leitores-escretores

Problema

- O problema dos leitores e escritores consiste em permitir que vários processos acessem simultaneamente a base dados para leitura, mas caso um processo esteja escrevendo na base de dados, nenhum outro processo poderá acessá-la nem para leitura nem para escrita.



Leitores-escretores

Solução

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

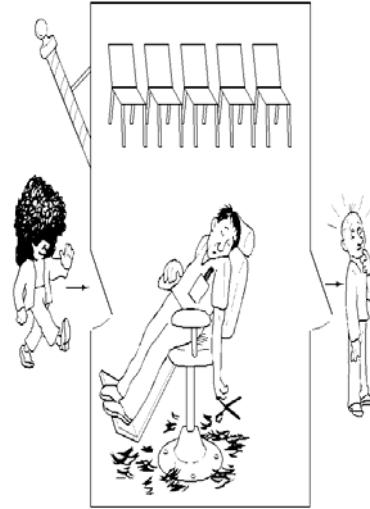
void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}
```

Barbeiro dorminhoco

Problema

- Em uma barbearia há um barbeiro, uma cadeira de barbeiro e **N** cadeiras para eventuais clientes esperarem a vez. Quando não há clientes, o barbeiro senta-se na cadeira do barbeiro e cai no sono. Quando chega um cliente, ele precisa acordar o barbeiro. Se outros clientes chegarem enquanto o barbeiro estiver cortando o cabelo de um cliente, eles se sentarão (se houver cadeiras vazias) ou sairão da barbearia (se todas as cadeiras estiverem ocupadas).



Barbeiro dorminhoco

Solução

```
#define CHAIRS 5
```

```
typedef int semaphore;
```

```
semaphore customers = 0;
```

```
semaphore barbers = 0;
```

```
semaphore mutex = 1;
```

```
int waiting = 0;
```

```
void barber(void)
```

```
{
    while (TRUE) {
        down(&customers);
        down(&mutex);
        waiting = waiting - 1;
        up(&barbers);
        up(&mutex);
        cut_hair();
    }
}
```

```
void customer(void)
```

```
{
    down(&mutex);
    if (waiting < CHAIRS) {
        waiting = waiting + 1;
        up(&customers);
        up(&mutex);
        down(&barbers);
        get_haircut();
    } else {
        up(&mutex);
    }
}
```



Monitores

- Mecanismos de sincronização estruturados de alto nível que tornam mais simples o desenvolvimento de aplicações concorrentes;
- Formados por procedimentos e variáveis encapsulados dentro de uma classe monitor;
- Implementação automática da exclusão mútua entre os métodos declarados em um monitor, pois somente pode estar executando um dos métodos do monitor em um determinado instante.

```
monitor example
  integer i;
  condition c;

  procedure producer();
  .
  .
  .
end;

  procedure consumer( );
  .
  .
  .
end;
end monitor;
```



Produtor-Consumidor Solução utilizando monitores

```
monitor ProducerConsumer
  condition full, empty;
  integer count;
  procedure insert(item: integer);
  begin
    if count = N then wait(full);
    insert_item(item);
    count := count + 1;
    if count = 1 then signal(empty)
  end;
  function remove: integer;
  begin
    if count = 0 then wait(empty);
    remove = remove_item;
    count := count - 1;
    if count = N - 1 then signal(full)
  end;
  count := 0;
end monitor;
```

```
procedure producer;
begin
  while true do
    begin
      item = produce_item;
      ProducerConsumer.insert(item)
    end
  end;
procedure consumer;
begin
  while true do
    begin
      item = ProducerConsumer.remove;
      consume_item(item)
    end
  end;
end;
```

