

Readers-Writers solution with concurrent reader access

Procedure reader

```
P(reader_mutex)
if readers = 0 then
  readers = readers + 1
  P(writer_mutex)
else
  readers = readers + 1
V(reader_mutex)
```

<read file>

```
P(reader_mutex)
readers = readers - 1
if readers == 0 then V(writer_mutex)
V(reader_mutex)
```

Procedure writer

```
P(writer_mutex)
<write file>
V(writer_mutex)
```

Readers-Writers with reader's priority

Procedure reader

```
P(reader_mutex)
if readers = 0 then
    readers = readers + 1
    P(writer_mutex)
else
    readers = readers + 1
V(reader_mutex)

<read file>

P(reader_mutex)
readers = readers - 1
if readers == 0 then V(writer_mutex)
V(reader_mutex)
```

Procedure writer

```
P(sr_mutex)
P(writer_mutex)

<write file>

V(writer_mutex)
V(sr_mutex)
```

Readers-Writers with reader's priority

Procedure reader

```
P(reader_mutex)
if readers = 0 then
  P(writer_mutex)
  readers = readers + 1
V(reader_mutex)

<read file>
```

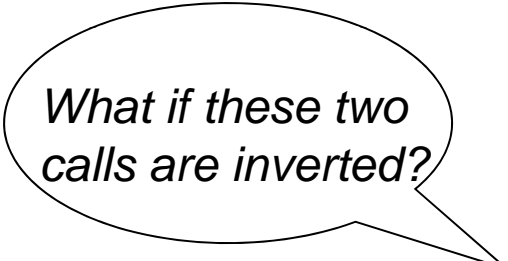
```
P(reader_mutex)
readers = readers - 1
if readers = 0 then V(writer_mutex)
V(reader_mutex)
```

Procedure writer

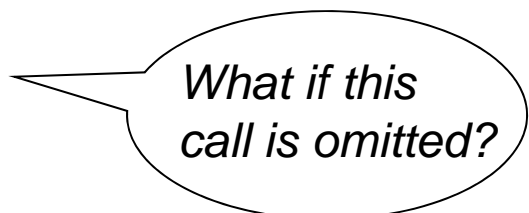
```
P(sr_mutex)
P(writer_mutex)

<write file>

V(writer_mutex)
V(sr_mutex)
```



What if these two calls are inverted?



What if this call is omitted?

Producer-Consumer: solution #1

Process producer

```
.  
. while count = N  
    ;  
    P(mutex)  
    count = count + 1  
    write(head_ptr)  
    head_ptr = (head_ptr + 1) mod N  
    V(mutex)  
.   
.
```

Process consumer

```
.  
. while count = 0  
    ;  
    P(mutex)  
    count = count - 1  
    read(tail_ptr)  
    tail_ptr = (tail_ptr + 1) mod N  
    V(mutex)  
.   
.
```

- Semaphore mutex ensures mutual exclusion in accessing the pool, however solution shown is not correct because variable *count* is not protected (for example two producers could enter when *count* = N-1)

Producer-Consumer: Correct Solution

Process producer

```
.  
.   
P(mutex)  
if count = N  
    then V(mutex); P(mutex_p); P(mutex)  
else  
    P(mutex_p) ;  
count = count + 1  
write(head_ptr)  
head_ptr = (head_ptr + 1) mod N  
V(mutex_c)  
V(mutex)  
.
```

Process consumer

```
.  
.   
P(mutex)  
if count = 0  
    then V(mutex); P(mutex_c); P(mutex)  
else  
    P(mutex_c) ;  
count = count - 1  
read(tail_ptr)  
tail_ptr = (tail_ptr + 1) mod N  
V(mutex_p)  
V(mutex)
```

- Initialize: $count = 0$; $mutex_c \doteq 0$; $mutex_p = N$;
- Assertions $count == mutex_c$; $count + mutex_p = N$

Producer-Consumer: another solution ??

Process producer

·
·

P(mutex)

P(mutex_p) ;

count = count + 1

write(head_ptr)

head_ptr = (head_ptr + 1) mod N

V(mutex_c)

V(mutex)

·
·

Process consumer

·
·

P(mutex)

P(mutex_c) ;

count = count - 1

read(tail_ptr)

tail_ptr = (tail_ptr + 1) mod N

V(mutex_p)

V(mutex)

·
·

- Initialize: count = 0; mutex_c = 0; mutex_p = N ;
- Assertions count == mutex_c ; count + mutex_p = N
- Does not work – DEADLOCK !!

Semaphore: pros and cons

- Pros:
 - no waste of resources due to busy waiting
 - flexible resource management using an initial value > 1
- Cons:
 - processes using semaphores must be aware of each other and coordinate respective use of semaphores
 - insertion of P and V calls is tricky and prone to errors
 - correctness of program using semaphores can be very hard to verify
 - do not scale up well - i.e. impractical for large scale use

Monitors: definition

- Monitors are abstract data types for encapsulating shared resources
- A monitor consists of:
 - shared objects and local variables,
 - a set of procedures
- Basic properties of the monitor
 - procedures are the only operations that can be performed on the resource and on the local variables
 - only one process at a time can be active (i.e. executing a procedure) within a monitor

Monitors: condition variables

- Condition variables are variables on which two operations are defined, *wait* and *signal*:
 - syntax: **<variable>.wait** and **<variable>.signal**
- They are used to delay and resume execution of processes calling monitor's procedures
- Condition variables are visible only from within monitor procedures

Semantic of *wait* and *signal*

- A queue is associated with each condition variable
 - *<variable>.queue* returns **true** if queue is not empty
- The *<variable>.wait* call suspends the calling process
 - calling process relinquishes control of the monitor
 - calling process is enqueued on the variable's queue
- The *<variable>.signal* call causes one waiting process to gain control of the monitor
 - it resume execution from where it left (i.e. right after the wait statement)
 - the calling process is enqueued on the *urgent* queue

Readers-Writers solution with readers' priority

```
procedure startRead;  
begin  
  if busy then OKtoread.wait;  
  readcount := readcount + 1;  
  OKtoread.signal;  
end startRead;
```

```
procedure endRead;  
begin  
  readcount := readcount - 1;  
  if readcount = 0 then OKtowrite.signal;  
end endRead;
```

```
procedure startWrite;  
begin  
  if busy OR readcount ≠ 0  
  then OKtowrite.wait;  
  busy := true;  
end startWrite;
```

```
procedure endWrite;  
begin  
  busy := false;  
  if OKtoread.queue  
  then OKtoread.signal  
  else OKtowrite.signal;  
end endWrite;
```