

Locks: pros and cons

- Pros:
 - simple and fast
 - ubiquitous: every processor has a test-and-set or equivalent operation
- Cons:
 - busy waiting is wasteful of resources (CPU cycles, memory bandwidth)

Semaphores - definition

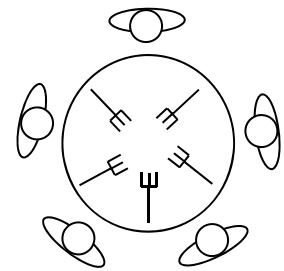
- Proposed by Dijkstra, it was the first high level constructs used to synchronize concurrent processes.
- A semaphore S is an integer variable on which two atomic operations are defined, $P(S)$ and $V(S)$, and with an associated queue.
- P and V semantic:

```
P(S) : if  $S \geq 1$  then  $S := S - 1$   
       else <block and enqueue the process>;
```

```
V(S) : if <some process is blocked on the queue> then  
       <unblock a process>  
       else  $S := S + 1$ ;
```

Other synchronization problems

- Semaphore can be used in other synchronization problems besides Mutual Exclusion
- The Producer-Consumer problem
 - a finite buffer pool is used to exchange messages between producer and consumer processes
- The Readers-Writers Problem
 - reader and writer processes accessing the same file
- The Dining Philosophers Problem
 - five philosophers competing for a pair of forks



Reader-Writers problem

- The shared resource is a file accessed by both reader and writer processes
- The synchronization constraints are:
 - readers should be able to concurrently access the file
 - only one writer at a time can access the file
 - readers and writers exclude each others
- Variants:
 - reader's priority: arriving readers have priority over waiting writers
 - writer's priority: writers have priority over waiting readers

Simple Readers-Writers solution

- The following scheme is very simple but does not allow concurrent reader access

Procedure reader

P(mutex)

<read file>

V(mutex)

Procedure writer

P(mutex)

<write file>

V(mutex)

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>
```

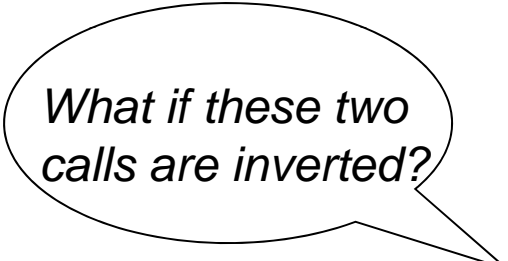
```
P(reader_mutex)
readers = readers - 1
if readers = 0 then V(writer_mutex)
V(reader_mutex)
```

Procedure writer

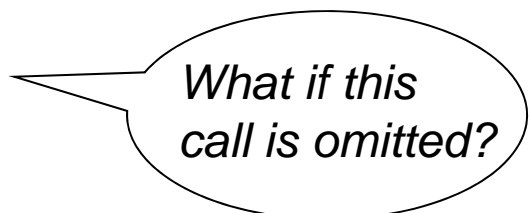
```
P(sr_mutex)
P(writer_mutex)

<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