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 ≥ 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;
```

Semaphores - properties

- The P operation may block a process, but V does not
- Two type of semaphores
  - binary: initial value is 1
  - resource counting: any initial value
- P and V are atomic operations

```
P(S): if S ≥ 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;
```

Example of use

Shared var mutex: **semaphore = 1;**

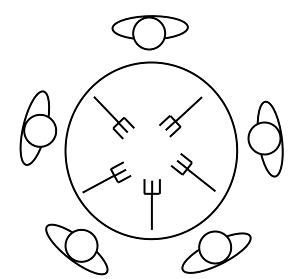
Process *i*

begin
.
.
P(mutex**);**
execute CS;
V(mutex**);**
.
.
End;

- Mutex vs. Lock?

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 forks



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)
- Would it work if switching the while loops and P(mutex)'s?

Would this work?

Process producer

.

.

P(mutex)

while count = N ;

count = count + 1

write(head_ptr)

head_ptr = (head_ptr + 1) mod N

V(mutex)

.

.

Process consumer

.

.

P(mutex)

while count = 0 ;

count = count - 1

read(tail_ptr)

tail_ptr = (tail_ptr + 1) mod N

V(mutex)

.

.

- Possible Deadlocks!

Producer-Consumer: Correct Solution

Process producer

.

.

P(mutex)

if count = N

  then V(mutex); P(sem_p); P(mutex)

else

  P(sem_p) ;

count = count + 1

write(head_ptr)

head_ptr = (head_ptr + 1) mod N

V(sem_c)

V(mutex)

.

.

Process consumer

.

.

P(mutex)

if count = 0

  then V(mutex); P(sem_c); P(mutex)

else

  P(sem_c) ;

count = count - 1

read(tail_ptr)

tail_ptr = (tail_ptr + 1) mod N

V(sem_p)

V(mutex)

.

.

- Initialize: count = 0; sem_c = 0; sem_p = N ;
- Invariants: count == sem_c ; sem_c + sem_p = N
- Really Correct?

Producer-Consumer: another solution ??

Process producer

.

.

P(mutex)

P(sem_p)

count = count + 1

write(head_ptr)

head_ptr = (head_ptr + 1) mod N

V(sem_c)

V(mutex)

.

.

Process consumer

.

.

P(mutex)

P(sem_c)

count = count - 1

read(tail_ptr)

tail_ptr = (tail_ptr + 1) mod N

V(sem_p)

V(mutex)

.

.

- Initialize: count = 0; sem_c = 0; sem_p = N ;
- Assertions count == sem_c ; sem_c + sem_p = N

- Does not work – DEADLOCK !!
- How can we solve the problem?

Quiz

- Using an exchange/swap instruction (atomic) to implement lock and unlock operations.
- Assuming the following semantics of a exchange/ swap instruction.

```
void swap (int *a, int *b)
{
        int tmp;
        tmp = *a;
        *a = *b;
        *b = tmp;
}
```