

The Concept and Use of Semaphores

Vlad Wojcik, Computer Science

"Every man, wherever he goes, is encompassed by a cloud of comforting convictions, which move with him like flies on a summer day"

(Bertrand Russell, Sceptical Essays, 1928, "Dreams and Facts")

DEFINITION:

Semaphore: A data structure, initialized at boot time of the machine, masquerading as a non – negative integer

PERMISSIBLE OPERATIONS:

Given a semaphore s , two **non-divisible** operations are defined:

```
signal(s)  // increments s by one
wait(s)    // decrements s by one as soon as it is possible
```

Notes:

```
signal(s)  is NOT equivalent to  s := s + 1
wait(s)    is NOT equivalent to  when (s > 0) s := s - 1
```

```
Value_of (s) = init(s) + number_of_signals(s)
               - number_of_successful_waits(s)
```

PURPOSES:

1. Enforcement of mutual exclusion
2. Synchronization (between loosely coupled processes)

ENFORCEMENT OF MUTUAL EXCLUSION:

Critical section: section of program code
not simultaneously available to several processes

Wrong (naïve) solution:

```
while (gate == closed) continue;
gate := closed;
// Critical section code goes here;
gate := open;
```

Correct solution using a semaphore named **mutex**, initialized to 1:

```
wait(mutex);
// Critical section code goes here;
signal(mutex);
```

Practical example: adding / removing items from a queue (**mutex** initialized to 1):

Adding process

```
.
.
wait(mutex);
add item to queue;
signal(mutex);
.
.
```

Removing process

```
.
.
wait(mutex);
remove item from queue;
signal(mutex);
.
.
```

SYNCHRONIZATION:

We have two processes A and B. We require that A should not proceed beyond point L1 until B reaches point L2. We use a semaphore **proceed** initialized to 0.

Code of A

```
.  
.   
L1 : wait(proceed) ;  
.   
.
```

Code of B

```
.   
.   
L2 : signal(proceed) ;  
.   
.
```

PRACTICAL EXAMPLE:

We have a pool of producer processes and another pool of consumer processes. Items of information created by producers are disposed of by consumers. The producers deposit their items in a buffer of capacity N. The consumers remove items in order to dispose of them.

Reasons for synchronization of access to the buffer (of capacity N, contents n):

- It is impossible to extract items if $n = 0$
- It is impossible to deposit items if $n = N$
- Buffer access is critical

Semaphores used: **mutex** initialized to 1
 space_available initialized to N
 item_available initialized to 0

Producer processes

```
.  
.  
repeat forever:  
begin  
    produce item;  
    wait(space_available);  
    wait(mutex);  
    deposit item in buffer;  
    signal(mutex);  
    signal(item_available);  
end
```

Consumer processes

```
.  
.  
repeat forever:  
begin  
    wait(item_available);  
    wait(mutex);  
    extract item from buffer;  
    signal(mutex);  
    signal(space_available);  
    consume item;  
end
```

A NASTY BUG CHALLENGE:

Find the bug in this solution:

Semaphores used: `mutex` initialized to 1
 `space_available` initialized to N
 `item_available` initialized to 0

Producer processes

```
.  
.
repeat forever:
begin
  produce item;
  wait(space_available);
  wait(mutex);
  deposit item in buffer;
  signal(mutex);
  signal(item_available);
end
```

Consumer processes

```
.  
.
repeat forever:
begin
  wait(mutex);
  wait(item_available);
  extract item from buffer;
  signal(mutex);
  signal(space_available);
  consume item;
end
```

GLOBAL SEMAPHORE:

A semaphore available to a number of processes.

Each process is allowed to perform both **wait** and **signal** operations on this semaphore.

TYPICAL USE: to protect mutually exclusive operations.

PRIVATE SEMAPHORE:

A semaphore available to a number of processes.

Each process is allowed to perform **signal** operations on this semaphore, but only one process is allowed to perform the **wait** operation.

TYPICAL USE: by processes wishing to check if they may proceed.

EXAMPLE:

Whenever a process has to decide if it can continue, the sequence of operations is:

```
.  
.   
.   
wait(mutex);  
Inspect relevant storage registers;  
if they show that the process can continue, then  
    perform signal(private_semaphore);  
signal(mutex);  
.   
.   
.   
wait(private_semaphore);
```

NOTE: **mutex** - global semaphore protecting the examination of registers (initially 1);
private_semaphore - initially 0.

EXAMPLE:

When a process reaches a stage where one or more other processes may have become free to proceed, the sequence of operations is:

```
.   
wait(mutex);  
Inspect and modify relevant storage registers;  
perform signal operations on the appropriate private semaphores;  
signal(mutex);  
. 
```


The semaphore system also formalizes the means whereby a process can safely perform "privileged operation(s)" on behalf of other process(es).

Case study: disk transfers:

Usually the access to the disk is a privilege reserved to the disk manager process. It has its own private semaphore **DM**, and there is a communication area in which the details of transfers required by client processes are placed. The disk manager can place there the feedback information as well.

This area may constitute a queue of requests of disk transfers. The global semaphore **Q** protects this queue. It is set initially to $n-1$, where n is the maximum number of requests that can be queued.

When a process wants a disk transfer to be performed on its behalf, the sequence of instructions is as follows:

```
.  
.   
.   
wait(Q);  
wait(mutex);  
record details of transfer on queue;  
signal(mutex);  
signal(DM);  
wait(private-semaphore);  
wait(mutex);  
read answer-back information from the communications area;  
signal(mutex);  
.   
.   
.
```

The sequence of operations of a disk manager process:

```
.  
.   
.   
START: wait(mutex);  
    read details of transfer from the queue;  
    pop-up the queue;  
    signal(mutex);  
    signal(Q);  
    perform the requested transfer to/from a disk;  
    wait(mutex);  
    record answer-back information in communications area;  
    signal(mutex);  
    signal(private_semaphore) -- of the client process;  
    wait(DM) -- on its own private semaphore;  
    goto START;
```

WARNING:

These examples illustrate the scheduling and synchronization problems only!