Semaphores.

Dijkstra, 1965. Semaphore is an synchronization object having two operations **P,V**.
Initialization:

```
      init(S, <number>);

P(S):
      if (S > 0)
        S--;
      else
        <wait S>;

V(S):
      if (<one or more processes are waiting S>)
        <free one process>;
      else
        S++;
```

P,S – atomic.
There are *binary* {0,1} semaphores and *counting* semaphores*, {0,1, 2,...}*.

Usage.

Critical section (mutual exclusion) implementation using binary semaphore.

```
init(S, 1);
P(S); < CS code>; V(S);
```

Waiting-signaling

```
init(S, FALSE);
```

`//. . .`
`V(S); // --> Signaling`
`//. . .`
	`P(S) // <-- Waiting`
	`//. . .`
	`//. . .`

P*roducer-consumer* problem.

The problem is how to implement data channel between producer and consumer processes using single shared variable or data buffer for data exchange.

1. Single shared variable.

```
init(S_consumed,TRUE);
init(S_produced,FALSE);
DATA d; // Shared
```

`// Producer`
`while(TRUE){`
`  DATA dLocal = produce();`
`  P(S_consumed);`
`  d = dLocal;`
`  V(S_produced);`
`}`
	`// Consumer`
	`while(TRUE){`
	`  P(S_produced);`
	`  DATA dLocal = d;`
	`  V(S_consumed);`
	`  consume(dLocal);`
	`}`

2. Implementation using size N buffer as a "ring".

```
init(S_consumed,N);
init(S_produced,0);
DATA d[N]; // Shared
```

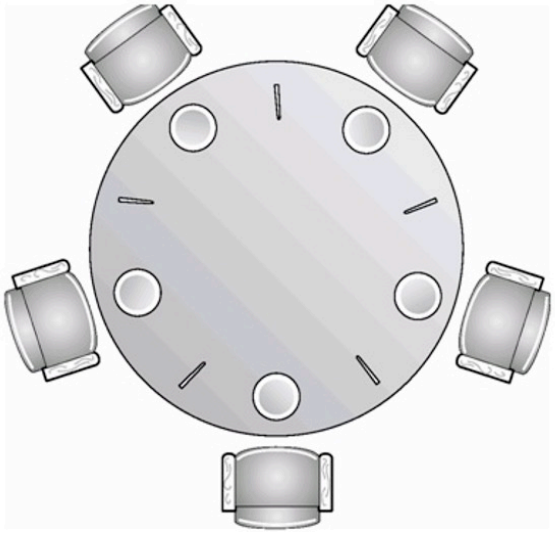
// Gamintojas	*// Vartotojas*

`int k = 0;`
`while(TRUE){`
`  DATA dLoc = produce();`
`  P(S_consumed);`
`  d[k] = dLoc;`
`  V(S_produced);`
`  k = (k+1) % N;`
`}`
	`int k = 0;`
	`while(TRUE){`
	`  P(S_produced);`
	`  DATA dLoc = d[k];`
	`  V(S_consumed);`
	`  k = (k+1) % N;`
	`  consume(duom);`
	`}`

Coordination access to resources

init(<number of available resources>

- **`P(S)`** – request resource;
- **`V(S)`** – release resource.



Clasical synchronization problems. Dinning philosophers.

[Wiki] Five philosophers sit at a round table with bowls of spaghetti. Forks are placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat spaghetti when he has both left and right forks. Each fork can be held by only one philosopher and so a philosopher can use the fork only if it is not being used by another philosopher. After he finishes eating, he needs to put down both forks so they become available to others. A philosopher can take the fork on his right or the one on his left as they become available, but cannot start eating before getting both of them.
Problem: avoid the deadlock (next solution is erroneous).

```
philosopher (){ // Incorrect!
  while(TRUE){
    think();
    grab_left_fork (); // P(sem_left)
    grab_right_fork(); // P(sem_right)
    eat();
    return_right_fork  (); // V(sem_right)
    return_left_fork   (); // V(sem_left)
  }}
```

Sleeping barber.

[Wiki] The barber has one barber chair and a waiting room with a number of chairs in it. When the barber finishes cutting a customer's hair, he dismisses the customer and then goes to the waiting room to see if there are other customers waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are no other customers waiting, he returns to his chair and sleeps in it.
Each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, then the customer wakes him up and sits in the chair. If the barber is cutting hair, then the customer goes to the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, then the customer leaves.

The above description should ensure that the shop functions correctly, with the barber cutting the hair of anyone who arrives until there are no more customers, and then sleeping until the next customer arrives.

Solution using semaphores.

```
init(SEM_clients, 0);
init(SEM_barber, 0);
init(SEM_mutex,   1);
int waiting = 0 ;
```

// Barber	*// Client*

`while(TRUE){`
`  P(SEM_clients);`
`  P(SEM_mutex);`
`  waiting--;`
`  V(SEM_barber);`
`  V(SEM_mutex);`
`  <cut the hair>;`
`}`
	`P(SEM_mutex)`
	`if (waiting >= places)`
	`  V(SEM_mutex); return}; //>>>`
	`waiting++;`
	`V(SEM_clients);`
	`V(SEM_mutex);`
	`P(SEM_barber);`
	`<sit in chair>;`

Critics of semaphores.

- The purpose (mutual exclusion, resource allocation, signaling) is not clear understandable from given context.
- They are closely related in the complex synchronization schemes.
- Semaphores might be a source of deadlock.

The conclusion: semaphores are low level synchronization objects.

Eventcount as semaphore alternative.

Reed (1977) & *Kanodia* (1979).

Eventcount: integer value (initial 0).

`advance(eventcount)`	Increment by 1 (atomic action);
`read(eventcount)`	Read the counter;
`await(eventcount, value)`	Wait until *counter* >= *value*.

Produced –consumer implementation using eventcounts.

```
// Global
const int N = 5;
eventcount in, out;
int buf[N];
```

```
// Producer
int i = 0;
while(TRUE){
   i++;
   await(out, i-N);
   buf[i%N] = produce();
   advance(in);
}
```

```
// Concumer
int i = 0;
while(TRUE){
   i++;
   await(in, i);
   consume(buf[i%N]);
   advance(out);
}
```