

# Concurrent Systems

*Nebenläufige Systeme*

## VII. Semaphore

Wolfgang Schröder-Preikschat

November 20, 2014



# Agenda

---

Preface

Fundamentals

- Classification

- Characteristics

Implementation

- Data Structures

- Functions

- Mutex

Summary



# Outline

---

Preface

Fundamentals

Classification

Characteristics

Implementation

Data Structures

Functions

Mutex

Summary



- discussion on **abstract concepts** as to unilateral and multilateral synchronisation, thus, partial and mutual exclusion
  - with the **general semaphore** as a measure that supports both
  - while the **binary semaphore** was/is intended to support the latter, only
- comprehensive differentiation of **semaphore** and **mutex**
  - in terms of the mutual exclusion aspect only, computer science folklore is right in stating disparities between the general variant and a mutex
  - but one have to be much more precise and argue with caution as far as the binary alternative is concerned:

## Hint (Methods v. Implementation/Object)

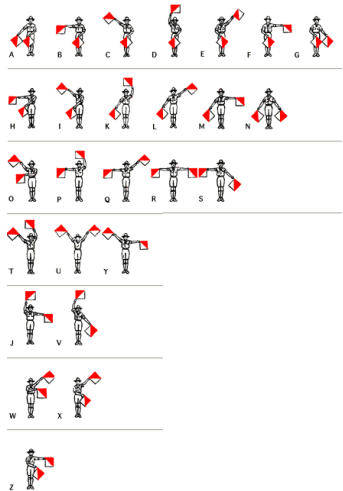
*A **binary semaphore** is a valid implementation of one of the many “mutex methods”, but not that restrictive as a “mutex object” need to be.*

- elaboration of various implementation aspects regarding both types of semaphore as well as mutex as an object





(Ger.) *Signalmast, Formsignal*



(Ger.) *Flaggensignal*



# Outline

---

Preface

**Fundamentals**

Classification

Characteristics

Implementation

Data Structures

Functions

Mutex

Summary



## Definition (Binary Semaphore)

*The semaphores are essentially non-negative integers; when only used to solve the mutual exclusion problem, the range of their values will even be restricted to "0" and "1". [2, p. 28]*

- jumping-off point for **sleeping lock** (Ger. *Schlafsperr*, [8, p. 9]) and, in particular, **mutex** (abbr. *mutual exclusion*)

## Definition (General Semaphore)

*It is the merit of [...] C. S. Scholten to have demonstrated a considerable field of applicability for semaphores that can also take on larger values. [2, p. 28]*

- also referred to as **counting semaphore** (Ger. *zählender Semaphor*)



# Elementary Operations

- insensitive to the distinction between binary and general semaphore is the definition of two **intrinsic primitives** [1]:

P abbr. for (Hol.) *prolaag*; a.k.a. *down*, *wait*, or *acquire*, resp.

- decreases<sup>1</sup> the value of the semaphore by 1:
  - i iff the resulting value would be non-negative [2, p. 29]
  - ii non-constraining [3, p. 345]
- blocks the process iff the value is or was, resp., 0 before decrease
  - blocking processes are put on a **waitlist** associated with each semaphore

V abbr. for (Hol.) *verhoog*; a.k.a. *up*, *signal*, or *release*, resp.

- increases<sup>1</sup> the value of the semaphore by 1
- as the case may be, unblocks a process blocked on the semaphore
  - which process becomes unblocked is to be regarded as unspecified

- each primitive needs to be considered as an **indivisible operation**

## Hint (Waitlist)

*The **queuing discipline** rivals with planning decisions of the process scheduler and, thus, may be the cause of critical **interference**.*

<sup>1</sup>This does not only mean subtraction or addition, resp., in arithmetical terms.



- **multilateral synchronisation** [5, p. 15] of interacting processes
  - the critical section is considered as a **non-preemptable reusable resource** that needs to be allocated indivisibly to a process to be usable correctly
  - in logical respect, the process having completed  $P$  on semaphore  $S$  is the only one being authorised to complete  $V$  on  $S$

```

1 semaphore_t mutex = {1};
2
3 {
4     P(&mutex);
5     /* critical section */
6     V(&mutex);
7 }
```

- default value is, normally, 1
  - block out only in the moment of a simultaneous process
  - allow full bent, else
- in case of a default value of 0
  - $V$  must come before  $P$

### Hint (Mutex (cf. p. 14/15))

A *mutex* is a **binary semaphore** that incorporates an **explicit check for authorisation** to release a critical section in the moment of  $V$ .



- **unilateral synchronisation** [5, p. 15] of interacting processes
  - used for **availability control** of entities of the following resource types:
    - i a **consumable resource** in the form of any data of any number
    - ii a **reusable resource** of limited number, e.g., a data store (buffer), any device
  - typical for, but not limited to, producer/consumer systems
- also as noted previously [5, p. 15], this art of synchronisation means:
  - logical** ■ coordination as indicated by a particular “role playing”
    - e.g., in order to proceed, a “data consumer” depends on the data to be made available by a “data producer”
  - conditional** ■ coordination as indicated by a condition for making progress
    - e.g., in order to proceed, a “data producer” depends on the store available for data handling
    - in the end, the data store will have to be deallocated and, thus, made available again by the “data consumer”
- from this it follows that  $P$  and  $V$  applied to the same semaphore  $S$  must have to be accomplishable by different processes, normally
  - which makes the big difference to a binary semaphore or mutex, resp.



```
1 semaphore_t data = {0};
2
3 void producer() {
4     for (;;) {
5         /* data released */
6         V(&data);
7     }
8 }
9
10 void consumer() {
11     for (;;) {
12         P(&data);
13         /* data acquired */
14     }
15 }
```

- usually, producer and consumer are different interacting processes
  - in case of one and the same process, the number of a completed  $V$  must exceed the number of a completed  $P$  in order to prevent deadlock
  - $\#V > \#P$ , which implies a path  $V \rightarrow P$  (i.e.,  $V$  “happens before”  $P$ )

- default value is 0
  - $P$  must block out only if there is no data
  - $V$  indicates more data
- calling sequence
  - $V$  must be actable independent of  $P$
  - in order to complete,  $P$  depends on  $V$
- ↪ beware of an **overflow** of the values margin



```
1 semaphore_t store = {N};
2
3 void producer() {
4     for (;;) {
5         P(&store);
6         /* store acquired */
7     }
8 }
9
10 void consumer() {
11     for (;;) {
12         /* store released */
13         V(&store);
14     }
15 }
```

- as to interacting processes in the line of producer and consumer, the same applies as mentioned before:  $\#V > \#P$
- in other cases:  $\#V \leq \#P$ , must be completed by the same process

- default value is  $N \geq 0$ 
    - $P$  must block out only if there is no store
    - $V$  indicates more store
  - calling sequence
    - $V$  must be actable independent of  $P$
    - in order to complete,  $P$  depends on  $V$
- ↪ beware of an **overflow** of the values margin



## Hint (Bounded Buffer)

*A means of managing an unlimited number of consumable resources on the basis of a limited number of reusable resources.*

```
1 semaphore_t data = {0}, store = {N};    /* N > 0 */
2 void producer() {                       10 void consumer() {
3     for (;;) {                           11     for (;;) {
4         P(&store);                        12         P(&data);
5         /* store acquired */             13         /* data acquired */
6         /* data released */             14         /* store released */
7         V(&data);                        15         V(&store);
8     }                                    16     }
9 }                                       17 }
```

- indisputable classic in cooperation and communication of processes
  - simply a merge of the semaphore use pattern discussed as before
  - **transverse application** of  $P$  and  $V$  to a pair of general semaphores



## Hint

Checking **authorisation** for release of a critical section in that very moment is improper for a general semaphore, optional for a binary semaphore, and may be demanded for a mutex (cf. p. 15).

- demanded
  - a **mutex object** ensures that the release of critical section CS will succeed only for the process having acquired CS
  - by extending a binary semaphore,  $P$  will have to record and  $V$  will have to check ownership of CS
- improper
  - $P$  and  $V$  on a **general semaphore** must be accomplishable in particular also by different processes
  - this is prevented by a **mutex object**—but not by a mutex
- optional
  - basically, a **binary semaphore** may be implemented by a general semaphore  $S$ , with  $S \leq 1 \Rightarrow$  never a **mutex object**
  - values  $S > 1$  must be prevented either by the use pattern or by the implementation of  $P$  and  $V$
- if **authorisation fails**, the process attempting to release CS should be aborted—in kernel mode, the computing system must be halted. . .



## Hint (Computer Science Folklore)

*A semaphore can be released by any process.*

- incomplete or rough, if not broad-bush, phrase that must be regarded with suspicion—one have to distinguish between semaphore types<sup>2</sup>
  - strictly, essence of this phrase is **requirement** for a general semaphore
  - strictly as well, it is merely an **option** for a binary semaphore
    - in logical respect, a binary semaphore cannot be released by any process
    - in physical respect, this however is not a must for any implementation

## Hint (Computer Science Folklore)

*A mutex can be released only by the process having it acquired.*

- a phrase that is slanted towards only one aspect as to the leastwise twofold non-uniform common understanding about a mutex:
  - i a category of **methods** for ensuring mutual exclusion *or*
  - ii the **implementation** of one of these methods in terms of an **object**<sup>2</sup>

<sup>2</sup>see also p. 36



- the **standby position** of a process within  $P$  is passive, normally
  - “blocks the” or “unblocks a”, resp. (cf. p. 8), process means rescheduling
  - if so, both may also entail context switching—“may” because:
    - $P$  – if no further process is ready to run, the **idle loop** becomes active
      - in that case, the blocking process likewise may fade to the **idle process**
      - thus, doing without a dedicated **idle-process instance** and context switch
    - $V$  – if there is a **waiting process**, it will be set “ready to run” (cf. [9, p. 28])
      - in that case, **priority violation**<sup>3</sup> must be prevented (scheduling discipline!)
      - thus, the current process may defer to a prior-ranking one: context switch
  - all this makes  $P$  and  $V$  programs of the operating system machine level
- $P$  and  $V$  relies on **process management** of the operating system
  - one have to put the current process asleep and get a sleeping process up
  - in functional terms, however,  $P$  and  $V$  need not be system calls
  - in non-functional terms,  $P$  and  $V$  should be close to the **scheduler**
    - by settling  $P$  and  $V$  in the address space of the operating-system kernel *or*
    - by making scheduler functions available through “strawweight” system calls

---

<sup>3</sup>If at least one of the processes on the waitlist is of higher-priority than the current process but will not become “ready to run” or allocated the processor.



# Waitlist Association

- in order to aid  $V$ , processes blocked by  $P$  at a semaphore are entered on a waitlist in either logical or physical means
  - logical
    - to block, a **blocked-on mark** is stored in the process descriptor
    - to unblock, a process-table walk looks for that mark
    - ↪ constant ( $P$ ) and variable but bounded above ( $V$ ) run-time
    - ↪ blocked-on mark is a “magic” address, no extra attributes
  - physical
    - to block, the process descriptor joins a **queue data structure**
    - to unblock, a process descriptor is removed from that structure
    - ↪ variable but bounded above ( $P$ ) and constant ( $V$ ) run-time
    - ↪ additional queue attribute of the semaphore data structure
- desirable is to have the waitlist queuing discipline in compliance with the process scheduling discipline: **freedom of interference**
  - a characteristic by means of which **priority violation** will be prevented
  - usually, this excludes straightforward queuing disciplines such as FCFS

## Hint (Process-Table Walk—Conformance to Scheduling)

*Part of the scheduler, lookup function to locate a process descriptor on the basis of the blocked-on mark as search key.*



- in the absence of simultaneous processes, the implementation of a semaphore could be as simple as follows:

```
1 void prolaag(semaphore_t *sema) {
2     if (!claim(sema))    /* at the moment, unavailable */
3         sleep(&sema->wand);
4 }
5
6 void verhoog(semaphore_t *sema) {
7     if (unban(sema))    /* as from now, available */
8         rouse(&sema->wand);
9 }
```

- whereat *claim* decreases and *unban* increases the value of the semaphore according to binary or general, resp., characteristic<sup>4</sup>
- but, assuming that the presence of simultaneous processes is possible, this implementation shows a **race condition**  $\rightsquigarrow$  **lost wakeup**
  - 3 ■ while going to sleep, i.e. being “sleepy”, the process gets delayed
  - 7–8 ■ but in good faith of a sleeper, the “sleepy” process may be missed

<sup>4</sup>The implementation of these helper functions will be revealed later.



- $P$  and  $V$  itself constitute a **critical section**, likewise, that must be protected in order to function correctly
  - protection should be constructed **per semaphore instance**, not  $P/V$

```
1 void prolaag(semaphore_t *sema) {
2     atomic *sema = {
3         if (!claim(sema))
4             sleep(&sema->wand);
5     }
6 }
7
8 void verhoog(semaphore_t *sema) {
9     atomic *sema = {
10        if (unban(sema))
11            rouse(&sema->wand);
12    }
13 }
```

- as a process will have to block inside a critical section, **deregulation of protection** is indispensable for the period the process is blocked

### Deadlock Prevention

Provided that protection of the critical section on the  $P$  side is not deregulated, the  $V$  side will never complete and, thus, will never cause unblocking of a process:

- the right location for deregulation is *sleep*
- after the process was marked sleeping



- protection of the  $P/V$  pair against simultaneous processes sharing a semaphore follows either the blocking or non-blocking paradigm
  - blocking
    - inhibit FLIH<sup>5</sup>, postpone SLIH<sup>5</sup>, or lock process
    - problem-specific construction of an **enter/leave** pair
    - ↪ coming right up next in this lecture (cf. p. 22ff.)
  - non-blocking
    - fall back on the elementary operations of the ISA level
    - problem-specific construction of  $P$  and  $V$
    - ↪ coming up as a case study in the context of LEC 10/11
- more detailed analysis of the “atomic” version of  $P$  reveals another problem: **overtaking** of an aroused process
  - upon return from *sleep* a formerly blocked process may complete  $P$  by mistake, joining a process in the critical section to be protected by  $P$
  - note that completion of  $V$  also opens the door for any process, not only for a process having been blocked at the semaphore
  - ↪ aroused processes will have to **retry claiming**: `if`  $\mapsto$  `while`
- not least, concurrency had to be constricted to no more than what is absolutely necessary: reflect on *claim/sleep* and *unban/rouse*

<sup>5</sup>abbr. for *first- or second-level interrupt handling*, resp.

# Outline

---

Preface

Fundamentals

Classification

Characteristics

Implementation

Data Structures

Functions

Mutex

Summary



# Semaphore Data Type

```
1 typedef volatile struct semaphore {
2     int gate;           /* value: binary or general */
3     wand_t wand;       /* protective shield */
4 } semaphore_t;
```

- purpose of “wand” (Ger. *Zauberstab*) is to **safeguard** the semaphore operations in various respect
  - i protect  $P$  and  $V$  against simultaneous processes
  - ii give leeway for protection variants (cf. p. 20)
- a wand that takes care of **mutual exclusion** techniques as presented in the previous lecture could be the following:

```
1 typedef volatile struct wand {
2     lock_t clue;       /* protects P or V, resp. */
3     event_t wait;     /* list of sleeping processes */
4 } wand_t;
```

→ becoming acquainted with other wands is content of future lectures...



```
1 void prolaag(semaphore_t *sema) {
2     enter(&sema->wand);      /* avert overlapped P or V */
3     while (!claim(sema))     /* acquire semaphore */
4         sleep(&sema->wand);  /* await wakeup signal */
5     leave(&sema->wand);      /* allow P or V */
6 }
7
8 void verhoog(semaphore_t *sema) {
9     enter(&sema->wand);      /* avert overlapped P or V */
10    if (unban(sema))          /* release semaphore */
11        rouse(&sema->wand);  /* cause wakeup signal */
12    else                        /* no sleeping process... */
13        leave(&sema->wand);  /* allow P or V */
14 }
```

- exercise caution in the analysis of these program statements:
  - 3–4 ■ takes care of the overtaking-problem as to aroused processes
  - 10–11 ■ in case of (i) logical waitlist and (ii) strict binary semaphore, the search for sleeping processes happens unconditionally
    - in that particular case, there is no direct indication of sleepers



# Acquire and Release Semaphore

- load/store-based implementation for a **binary semaphore**:

```
1 inline bool claim(semaphore_t *sema) {
2     return (sema->gate == 0) ? false : (sema->gate = 1);
3 }
4
5 inline bool unban(semaphore_t *sema) {
6     return !(sema->gate = 0);
7 }
```

- enumerator-based implementation for a **general semaphore**:

```
1 inline bool claim(semaphore_t *sema) {
2     return sema->gate-- > 0;
3 }
4
5 inline bool unban(semaphore_t *sema) {
6     return (sema->gate++ < 0);
7 }
```

- note that both variants are sensitive to simultaneous processes
  - use within a safeguarded program section is assumed...



```

1 inline void sleep(wand_t *wand) {
2     catch(&wand->wait); /* disclose process to V */
3     leave(wand);        /* allow P or V */
4     coast();           /* take a break */
5     enter(wand);       /* apply for return to P */
6 }
7
8 inline void rouse(wand_t *wand) {
9     leave(wand);        /* allow P or V */
10    cause(&wand->wait); /* signal end of break */
11 }

```

- constrict concurrency to no more than what is absolutely necessary:
  - 2 ■ endorse interest of the current process of upcoming dormancy
  - 3 ■ soon dormant process was made known, deregulate  $P$  safeguard ←
  - 4 ■ transition to dormant state: rescheduling, context switch or idleness
  - 5 ■ apply for return to safeguarded  $P$
  - 9 ■ dormant processes could be available, deregulate  $V$  safeguard ←
  - 10 ■ annulment of dormant state: rescheduling, context switch

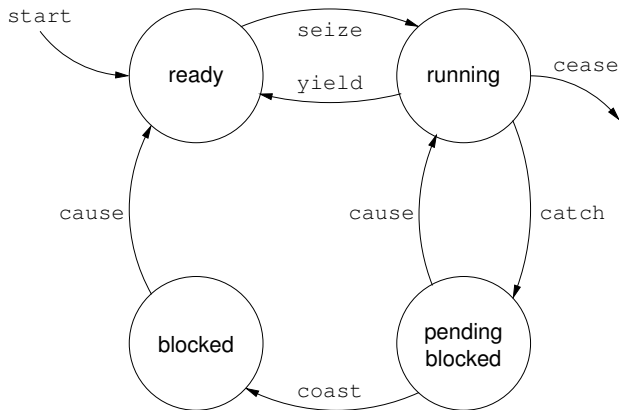


- catch* ■ exists in two variants, depending on the waitlist model (cf. p 17):
- i store of a blocked-on mark in the process descriptor *or*
  - ii enqueue of the process descriptor into a queue data structure
- variant (i) writes to an own data structure of the current process, while variant (ii) manipulates a shared data structure
- signals upcoming blocking (dormancy) of the registered process
- coast* ■ blocks the current process, reschedules the processor, and either performs a context switch or runs through the idle loop
- manipulates a shared data structure (ready list)
  - performs the queuing function of the queue-based *catch*
- eventually returns when the blocking condition was nullified
- cause* ■ unblocks the next registered process, if any, found by means of a (i) process-table walk or (ii) dequeue operation
- manipulates a shared data structure (ready list)
- if need be, the current process defers to a prior-ranking process

### Hint (Idle State (cf. p. 16 and p. 37))

*The last process blocked may find itself on the ready list. Same may happen to the “sleepy process” as coast runs deregulated to P/V.*





- **ready** ↔ **running** ■ scheduler
- blocked** → **ready** ■ iff *effective signalling* ( $V$ ), i.e., waiting process
- running** ↔ **pending** ■ doze ( $P \rightarrow$ ), *effective signalling* ( $\leftarrow V$ )
- pending** → **blocked** ■ deep sleep ( $P$ ), no overlapping  $V$



- as there is no single solution to protect  $P$  and  $V$  adequately, the wand attribute symbolises intention to application orientation
  - depending on the mode of operation or use case, the wand acts differently
  - assuming that processing elements are not multiplexed [7, p. 5], then:

```
1 inline void enter(wand_t *wand) {
2     lock(&wand->clue);
3 }
4
5 inline void leave(wand_t *wand) {
6     unlock(&wand->clue);
7 }
```

- wand capability depends on the “type of exclusion” in relation to the required characteristics of the operating system machine level:
  - partial** ■ processor **multiplexing**  $\rightsquigarrow$  interrupt control
  - mutual** ■ processor **multiplication**  $\rightsquigarrow$  process lock, see example above
- combination of both is optional, not mandatory, and problem-specific
  - depends on the degree of parallelism (a) allowed for by the application use case and (b) made possible by the ISA level



```

1      ...
2      movl  16(%esp), %edi
3      leal  4(%edi), %esi
4      jmp   LBB0_2
5  LBB0_1:
6      movl  _life, %eax
7      movl  %esi, 4(%eax)
8      movl  %esi, (%esp)
9      calll _unlock
10     calll _coast
11  LBB0_2:
12     movl  %esi, (%esp)
13     calll _lock
14     cmpl  $0, (%edi)
15     je    LBB0_1
16     movl  $1, (%edi)
17     movl  %esi, (%esp)
18     calll _unlock
19     ...

```

- let the sequence of instructions within  $P$  be as follows:
    - 2 ■ point at semaphore
    - 3 ■ point at lock structure
    - address is blocked-on mark
    - 12–13 ■ apply for  $P$  protection
    - 14–15 ■ check binary semaphore  $S_b$
    - 16–18 ■ unoccupied, take  $S_b$
    - quit  $P$  protection, done
    - 5 ■ occupied,  $S_b$  already taken
    - 6 ■ point at process structure
    - 7 ■ define blocked-on mark
    - 8–9 ■ deregulate  $P$  protection
    - 10 ■ fall asleep, dream about  $V$
  - locking overhead when unoccupied
    - net worth of about 5 instructions ☹
- ↪ non-blocking synchronisation ☺

<sup>6</sup>Take a sledgehammer to crack a nut...



- given the concept of a binary semaphore, implementation of a **mutex** is straightforward and, absolutely, no black magic:
  - a mutex data structure is composed of two parts:
    - i a binary semaphore used to actually protect the critical section *and*
    - ii a handle that uniquely identifies the process having acquired the mutex<sup>7</sup>
  - given such a structure, let the following two functions be defined:
    - acquire** – performs the  $P$  and registers the current process as owner
    - release** – conditionally unregisters the owner and performs the  $V$ 
      - in case of a wrong owner, the current process or kernel, resp., panics
- a corresponding **data type** may be laid out as follows:

```
1 typedef volatile struct mutex {
2     semaphore_t sema;    /* binary semaphore */
3     process_t *link;    /* owning process or 0 */
4 } mutex_t;
```

---

<sup>7</sup>At kernel level, the handle is the pointer to the process descriptor of the process instance. At user level, it is the process identification.



## Acquire and Release Mutex

```
1 extern void panic(char*) __attribute__((noreturn));
2
3 void acquire(mutex_t *mutex) {
4     P(&mutex->sema);          /* lockout */
5     mutex->link = being(ONESELF); /* register owner */
6 }
7
8 void release(mutex_t *mutex) {
9     if (mutex->link != being(ONESELF)) /* it's not me! */
10        panic("unauthorised release of mutex");
11
12     mutex->link = 0;          /* deregister owner */
13     V(&mutex->sema);        /* unblock */
14 }
```

- release of a mutex by an **unauthorised process** is a **serious matter**
  - presumably, the non-sequential program contains a **software fault** (bug)
  - returning an error code is no option, as one cannot rely on error checking
  - any other than “raising a non-maskable exception” is a botch job...



# Outline

---

Preface

Fundamentals

Classification

Characteristics

Implementation

Data Structures

Functions

Mutex

Summary



- fundamental concept for cooperation and communication
  - binary and general/counting semaphore, intrinsic primitives  $P$  and  $V$
  - correlation to unilateral and multilateral synchronisation
  - differentiation as to mutex (methods v. implementation/object):

## Hint

*A **binary semaphore** is a valid implementation of one of the many “mutex methods”, but not that restrictive as a “mutex object” need to be.*

- hierarchic placement at operating system machine level
- characteristics important in functional and non-functional terms
  - logical or physical waitlist, conformance to the scheduling discipline
  - deregulation of the protection of  $P$  against simultaneous processes
  - further shallows such as overtaking of unblocked processes in  $P$ :

## Hint

*Constrict concurrency to no more than what is absolutely necessary.*

- not least, basic approaches and sketches of an implementation. . .



# Reference List I

---

- [1] DIJKSTRA, E. W.:  
Over seinpalen / Technische Universiteit Eindhoven.  
Eindhoven, The Netherlands, 1964 ca. (EWD-74). –  
Manuskript. –  
(dt.) Über Signalmasten
- [2] DIJKSTRA, E. W.:  
Cooperating Sequential Processes / Technische Universiteit Eindhoven.  
Eindhoven, The Netherlands, 1965 (EWD-123). –  
Forschungsbericht. –  
(Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)
- [3] DIJKSTRA, E. W.:  
The Structure of the “THE”-Multiprogramming System.  
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346
- [4] PARNAS, D. L.:  
Some Hypothesis About the “Uses” Hierarchy for Operating Systems / TH  
Darmstadt, Fachbereich Informatik.  
1976 (BSI 76/1). –  
Forschungsbericht



- [5] SCHRÖDER-PREIKSCHAT, W. :  
**Concurrency.**  
In: [6], Kapitel 2
  
- [6] SCHRÖDER-PREIKSCHAT, W. ; LEHRSTUHL INFORMATIK 4 (Hrsg.):  
**Concurrent Systems.**  
FAU Erlangen-Nürnberg, 2014 (Lecture Slides)
  
- [7] SCHRÖDER-PREIKSCHAT, W. :  
**Introduction.**  
In: [6], Kapitel 1
  
- [8] SCHRÖDER-PREIKSCHAT, W. :  
**Locks.**  
In: [6], Kapitel 6
  
- [9] SCHRÖDER-PREIKSCHAT, W. :  
**Processes.**  
In: [6], Kapitel 3



## Commonalities and differences as to their possible **internal states**.

- general semaphore  $S_g$ :
  - positive ■  $N > 0$  processes will complete  $P(S_g)$  without blocking
  - zero ■  $P(S_g)$  will block the running process on the waitlist of  $S_g$
  - negative ■  $P(S_g)$  will block the running process on the waitlist of  $S_g$ 
    - $|N|$  processes are blocked on the waitlist of  $S_g$
- binary semaphore  $S_b$ :
  - not taken ■ exactly one process will complete  $P(S_b)$  without blocking
    - the very process becomes **logical owner** of  $S_b$
  - taken ■  $P(S_b)$  will block the running process on the waitlist of  $S_b$ 
    - $V(S_b)$  should be performed only by the logical owner of  $S_b$
- mutex object  $M$ : let  $A$  be *acquire* and let  $R$  be *release*
  - not owned ■ exactly one process will complete  $A(M)$  without blocking
    - the very process becomes **physical owner** of  $M$
  - owned ■  $A(M)$  will block the running process on the waitlist of  $M$ 
    - $R(M)$  can succeed only for the physical owner of  $M$



- principle pattern of a scheduler function to block a process
  - called by *coast* (cf. p.25) and other functions to pause computation

```
1 void block() {
2     process_t *next, *self = being(ONESELF);
3
4     while (!(next = elect(hoard(READY))))
5         relax();                /* no ready to run... */
6
7     if (next != self) {         /* must relinquish */
8         self->state = BLOCKED; /* vacate processor */
9         seize(next);          /* resume elected */
10    }
11    self->state = RUNNING;      /* occupy processor */
12 }
```

- 4 ■ choose next process to be dispatched to the processor
- 5 ■ ready list is empty, so the running process fades to the idle process
- 7 ■ as the case may be, the running process may be allowed to continue:
  - i the idle/running process found itself ready-to-run on the ready list *or*
  - ii the running process, sent to sleep due to  $P$ , was roused due to  $V$  (p.25)

