# Concurrent Systems

*Nebenläufige Systeme*

## VII. Semaphore

Wolfgang Schröder-Preikschat

December 13, 2016

# Agenda

# Outline

# Subject Matter

- 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/Entity)**

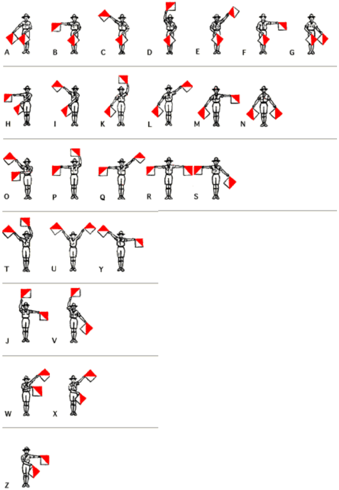
*A **binary semaphore** is a valid implementation of one of the many "mutex methods", but not that restrictive as a "mutex entity" need to be.*

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

# Colloquialism

(Gr.) *sēma·pherein*, (Ger.) *Gemeinsprache*, *Redensart*

(Ger.) *Signalmast*, *Formsignal*




(Ger.) *Flaggensignal*

# Outline

# Concept for Cooperation and Communication

## Definition (Binary Semaphor)

*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. *Schlafsperre*, [8, p. 9]) and, in particular, **mutex** (abbr. *mutual exclusion*)

## Definition (General Semaphor)

*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[1] 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[1] 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**.*

[1] This does not only mean subtraction or addition, resp., in arithmetical terms.

# Binary Semaphore

- **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 $S$

```
semaphore_t mutex = {1};

{
    P(&mutex);
    /* critical section */
    V(&mutex);
}
```

- 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 $V$.*

# General Semaphore

- **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.

# Consumable Resource



```c
semaphore_t data = {0};

void producer() {
    for (;;) {
        /* data released */
        V(&data);
    }
}

void consumer() {
    for (;;) {
        P(&data);
        /* data acquired */
    }
}
```

- 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

- usually, producer and consumer are <u>different</u> 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” $P$)

# Reusable Resource

```c
semaphore_t store = {N};

void producer() {
    for (;;) {
        P(&store);
        /* store acquired */
    }
}

void consumer() {
    for (;;) {
        /* store released */
        V(&store);
    }
}
```

- 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 $V$

  $\hookrightarrow$ beware of an **overflow** of the values margin

- 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

# Availability Control in Practice

## Hint (Bounded Buffer)

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

```
semaphore_t data = {0}, store = {N};    /* N > 0 */

void producer() {
  for (;;) {
    P(&store);
    /* store acquired */
    /* data released */
    V(&data);
  }
}

void consumer() {
  for (;;) {
    P(&data);
    /* data acquired */
    /* store released */
    V(&store);
  }
}
```

- 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

# Semaphore v. Mutex I

## 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 entity** 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 entity—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 entity
- 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...

# Semaphore v. Mutex II

## 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[2]
  - 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 may not 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 **entity**[2]

[2]see also p. 38

# Hierarchic Placement

- 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***[3] 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

[3] 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.*

# Indivisible Operation I



- 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[4]
- but, assuming that the presence of simultaneous processes is possible, this implementation shows a **race condition** $\leadsto$ ***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

[4] The implementation of these helper functions will be revealed later.

# Indivisible Operation II



- *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 $P/V$

```
void prolaag(semaphore_t *sema) {
    atomic *sema = {
        if (!claim(sema))
            sleep(&sema->wand);
    }
}

void verhoog(semaphore_t *sema) {
    atomic *sema = {
        if (unban(sema))
            rouse(&sema->wand);
    }
}
```

**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

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

# Shallows

- protection of the $P/V$ pair against simultaneous processes sharing a semaphore follows either the blocking or non-blocking paradigm

blocking
- inhibit FLIH[5], postpone SLIH[5], or lock process
- problem-specific construction of an ***enter*/*leave*** pair

↪ coming right up next in this lecture (cf. p. 22*ff.*)

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 $P$
  - this is because completion of $V$ also opens the door for any process, not only for a process having been blocked at the semaphore

↪ if applicable, aroused processes 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*

[5] abbr. for *first-* or *second-level interrupt handling*, resp.

# Outline

# Semaphore Data Type

```
typedef volatile struct semaphore {
    int gate;          /* value: binary or general */
    wand_t wand;       /* protective shield */
} 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 by means of locks [8], for example, could be the following:

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

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

# $P$ and $V$ Safeguarded



```
1  void prolaag(semaphore_t *sema) {
2      enter(&sema->wand);       /* avert overlapped P or V */
3      lodge(sema);              /* raise claim to proceed */
4      when (!avail(sema))       /* check for process delay */
5          sleep(&sema->wand);   /* await wakeup signal */
6      leave(&sema->wand);       /* allow P or V */
7  }
8  
9  void verhoog(semaphore_t *sema) {
10     enter(&sema->wand);       /* avert overlapped P or V */
11     if (unban(sema))          /* release semaphore */
12         rouse(&sema->wand);   /* cause wakeup signal */
13     else                      /* no sleeping process... */
14         leave(&sema->wand);   /* allow P or V */
15 }
```

- exercise caution in the analysis of these program statements:
  - 4 — if applicable, "*when*" takes care of overtaking processes
  - 11–12 — if applicable, search for sleepers happens unconditionally
    - in case of (i) logical waitlist and (ii) strict binary semaphore

# Acquire and Release Semaphore I

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

```
inline int lodge(semaphore_t *sema) {
    return 42;
}

inline bool avail(semaphore_t *sema) {
    return (sema->gate == 0) ? false : !(sema->gate = 0);
}

inline bool unban(semaphore_t *sema) {
    return (sema->gate = 1) && exist(&sema->wand);
}
```

  - note that the semaphore value alone shows no indication of processes that potentially await a reveille (Ger. *Wecksignal*) as to this very semaphore
  - only an explicit waitlist scan sheds light on that $\leadsto$ *exist*

- also note the persisting sensitivity to simultaneous processes: *avail*
  - use within a safeguarded program section is assumed...

# Acquire and Release Semaphore II

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

```
inline int lodge(semaphore_t *sema) {
    return sema->gate--;
}

inline bool avail(semaphore_t *sema) {
    return sema->gate >= 0;
}

inline bool unban(semaphore_t *sema) {
    return (sema->gate++ < 0);
}
```

  - note that the absolute value of a “negative semaphore” gives the number of processes awaiting a reveille as to this very semaphore
  - thus, there is no need for an explicit waitlist scan ☺

- also note the persisting sensitivity to simultaneous processes: --/++
  - use within a safeguarded program section is assumed...

# Overtaking Lane: Semaphore Handicap (cf. p. 20)

- in contrast to the shown general semaphore, a roused process has to **recheck his waiting condition** in $P$

- **general semaphore**
  - overtaking impossible
    - $gate \leq 0$ when a process aroused
    - rival process in $P$ causes $gate < 0$
    - ↪ will be forced to *sleep*
    - ↪ aroused process may proceed
  - `#define when if`
    - susceptible to erroneous rouse

- **binary semaphore**
  - overtaking possible
    - $gate = 1$ when a process aroused
    - rival process in $P$ causes $gate = 0$
    - ↪ is allowed to continue ☹
    - ↪ aroused process has to wait
  - `#define when while`
    - unsusceptible to erroneous rouse

### Hint (erroneous rouse)

*Caused by misuse of $V$ or by forced and uncontrolled unblocking of a process that went to sleep in $P$. Both are programming errors: the former at (semaphore) application level, the latter at system level.*

# Special Process Management



```
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

# General Process Management

*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
- signalises upcoming blocking (dormancy) of the registered process

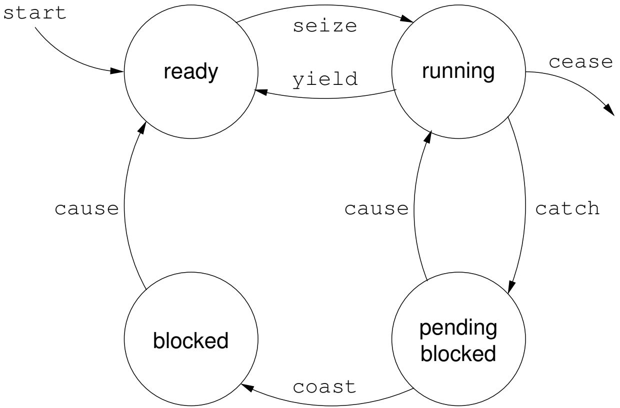
*coast*
- blocks the current process, reschedules the processor, and either performs a context switch or runs through the idle loop
  - manipulates a shared data structure (ready list)
  - performs the queuing function of the queue-based *catch*
- eventually returns when the blocking condition was nullified

*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. 39))**

*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.*

# Process States and State Transitions cf. [9, p. 27]




- 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 $V$

# Semaphore Gatekeeper

(Ger.) *Schrankenwärter, Türhüter*

- 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:

```
inline void enter(wand_t *wand) {
    lock(&wand->clue);
}

inline void leave(wand_t *wand) {
    unlock(&wand->clue);
}
```

- 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

# Specialisation of a Binary Semaphore



- 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[6]
  - 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:

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

[6] 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

```
extern void panic(char*) __attribute__ ((noreturn));

void acquire(mutex_t *mutex) {
    P(&mutex->sema);            /* lockout */
    mutex->link = being(ONESELF);   /* register owner */
}

void release(mutex_t *mutex) {
    if (mutex->link != being(ONESELF)) /* it's not me! */
        panic("unauthorised release of mutex");

    mutex->link = 0;            /* deregister owner */
    V(&mutex->sema);            /* unblock */
}
```

- 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

# Résumé

- 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/entity):

**Hint**

*A **binary semaphore** is a valid implementation of one of the many "mutex methods", but not that restrictive as a "mutex entity" 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

# Reference List II


[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

# Extent of Critical Section



```
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 ☺

[7] Take a sledgehammer to crack a nut...

# Semaphore v. Mutex III

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 entity $M$: 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 $M$

# Idle State

cf. p. 28

- principle pattern of a scheduler function to block a process
  - called by *coast* (cf. p. 27) 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. 27)