Concurrent Systems

*Nebenläufige Systeme*

VII. Semaphore

Wolfgang Schröder-Preikschat

November 25, 2015
Agenda

Preface

Fundamentals
  Classification
  Characteristics

Implementation
  Data Structures
  Functions
  Mutex

Summary
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
Colloquialism

(Gr.) sēma-pherein, (Ger.) Gemeinsprache, Redensart

(Ger.) Signalmast, Formsignal

(Ger.) Flaggensignal
Outline

Preface

Fundamentals
  Classification
  Characteristics

Implementation
  Data Structures
  Functions
  Mutex

Summary
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. Schlafsperrre, [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 $S$

```c
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 $V$.  

© wosch  
CS (WS 2015, LEC 7)  
Fundamentals – Classification
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.
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 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 → 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 $V$
- 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
A means of managing an unlimited number of consumable resources on the basis of a limited number of reusable resources.

```c
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
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...
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:
  - 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:
  1. a category of methods for ensuring mutual exclusion or
  2. the implementation of one of these methods in terms of an object

\[\text{see also p. 36}\]
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**\(^3\) 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.
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
  \( \leftrightarrow \) constant \((P)\) and variable but bounded above \((V)\) run-time
  \( \leftrightarrow \) 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
  \( \leftrightarrow \) variable but bounded above \((P)\) and constant \((V)\) run-time
  \( \leftrightarrow \) 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:

```c
void prolaag(semaphore_t *sema) {
    if (!claim(sema))  /* at the moment, unavailable */
        sleep(&sema->wand);
}

void verhoog(semaphore_t *sema) {
    if (unban(sema))   /* as from now, available */
        rouse(&sema->wand);
}
```

whereat `claim` decreases and `unban` increases the value of the semaphore according to binary or general, resp., characteristic

but, assuming that the presence of simultaneous processes is possible, this implementation shows a race condition \(\sim\) *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.
$P$ 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 $P/V$

```c
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

Implementation

- 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. 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 \(\Rightarrow\) 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.
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 as presented in the previous lecture could be the following:

```c
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...
void prolaag(semaphore_t *sema) {
    enter(&sema->wand);       /* avert overlapped P or V */
    while (!claim(sema))     /* acquire semaphore */
        sleep(&sema->wand);   /* await wakeup signal */
    leave(&sema->wand);      /* allow P or V */
}

void verhoog(semaphore_t *sema) {
    enter(&sema->wand);       /* avert overlapped P or V */
    if (unban(sema))          /* release semaphore */
        rouse(&sema->wand);   /* cause wakeup signal */
    else                      /* no sleeping process... */
        leave(&sema->wand);   /* allow P or V */
}

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:

```c
inline bool claim(semaphore_t *sema) {
    return (sema->gate == 0) ? false : (sema->gate = 1);
}

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

■ enumerator-based implementation for a general semaphore:

```c
inline bool claim(semaphore_t *sema) {
    return sema->gate-- > 0;
}

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

■ note that both variants are sensitive to simultaneous processes
  ■ use within a safeguarded program section is assumed…
Special Process Management

Prevent Lost Wakeup

```c
inline void sleep(wand_t *wand) {
    catch(&wand->wait); /* disclose process to V */
    leave(wand);       /* allow P or V */
    coast();           /* take a break */
    enter(wand);       /* apply for return to P */
}

inline void rouse(wand_t *wand) {
    leave(wand);       /* allow P or V */
    cause(&wand->wait); /* signal end of break */
}
```

constrict concurrency to no more than what is absolutely necessary:

- endorse interest of the current process of upcoming dormancy
- soon dormant process was made known, deregulate $P$ safeguard
- transition to dormant state: rescheduling, context switch or idleness
- apply for return to safeguarded $P$
- dormant processes could be available, deregulate $V$ safeguard
- annulment of dormant state: rescheduling, context switch
**General Process Management**

**Event Handling**

- **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. 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.
Process States and State Transitions

cf. [9, p. 27]

- start
- ready ↔ running
- blocked → ready
- running ↔ pending
- pending → blocked

- scheduler
- iff *effective signalling* ($V$), i.e., waiting process
- doze ($P \rightarrow$), *effective signalling* ($\leftarrow V$)
- deep sleep ($P$), no overlapping $V$

© wosch  
CS (WS 2015, LEC 7)  
Implementation – Functions
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:

```c
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** $\sim$ interrupt control
- mutual processor **multiplication** $\sim$ 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
let the sequence of instructions within $P$ be as follows:

1. point at semaphore
2. point at lock structure
3. address is blocked-on mark
4. apply for $P$ protection
5. occupied, $S_b$ already taken
6. point at process structure
7. define blocked-on mark
8. deregulate $P$ protection
9. fall asleep, dream about $V$
10. locking overhead when unoccupied
11. net worth of about 5 instructions

non-blocking synchronisation

---

6 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

- 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;
```

---

7 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

```c
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

Preface

Fundamentals
   Classification
   Characteristics

Implementation
   Data Structures
   Functions
   Mutex

Summary
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/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...
[1] **DIJKSTRA, E. W.**:
Over seinpalen / Technische Universiteit Eindhoven.
Manuskript. –
(dt.) Über Signalmasten

[2] **DIJKSTRA, E. W.**:
Cooperating Sequential Processes / Technische Universität Eindhoven.
Forschungsbericht. –

[3] **DIJKSTRA, E. W.**:
The Structure of the “THE”-Multiprogramming System.
In: *Communications of the ACM* 11 (1968), Mai, Nr. 5, S. 341–346

Some Hypothesis About the “Uses” Hierarchy for Operating Systems / TH Darmstadt, Fachbereich Informatik.
1976 (BSI 76/1). –
Forschungsbericht
In: [6], Kapitel  2

FAU Erlangen-Nürnberg, 2014 (Lecture Slides)

In: [6], Kapitel  1

In: [6], Kapitel  6

In: [6], Kapitel  3
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 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 $M$
Idle State

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

```c
void block() {
    process_t *next, *self = being(ONESELF);
    while (!(next = elect(hoard(READY))))
        relax(); /* no ready to run... */
    if (next != self) {
        /* must relinquish */
        self->state = BLOCKED; /* vacate processor */
        seize(next);
        /* resume elected */
    }
    self->state = RUNNING; /* occupy processor */
}
```

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)