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
Concept for Cooperation and Communication

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. Schlaf sperre, [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), to control resource allocation in a quantitive measure

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.
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 \( P \)

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

Thu Oct 23 14:45:50 CEST 2019

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

unilateral synchronisation [5, p. 15] of interacting processes

- used for availability control of entities of the following resource types:
  - a consumable resource in the form of any data of any number
  - 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

- 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.

Consumer Resource Availability Control

```
semaphore_t data = {0};
void producer() {
  for (;;) {
    /* data released */
    V(&data);
  }
}
void consumer() {
  for (;;) {
    P(&data);
    /* data acquired */
  }
}
```

- 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” \( P \))

Reusable Resource Availability Control

```
semaphore_t store = {N};
void producer() {
  for (;;) {
    P(&store);
    /* store acquired */
  }
}
void consumer() {
  for (;;) {
    P(&store);
    /* store released */
    V(&store);
  }
}
```

- 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
basically, a P

if there is a waiting process, it will be set “ready to run” (cf. [9, p.28])

if no further process is ready to run, the

V(& data );
/* data released */

P(& store );
/* data acquired */

for

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

for release of a critical section in that very

improper

P and V on a general semaphore must be accomplishable

{see also p.38

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 I

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).

Semaphore v. Mutex II

A semaphore can be released by any process.

incomplete or rough, if not broad-brush, 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 may not be released by any process

– in physical respect, this however is not a must for any implementation

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

3see also p. 38

© wosch, thoenig CS (WS 2019/20, LEC 7) Fundamentals–Classification


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 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
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
  - 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 queueing 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 queueing disciplines such as FCFS

---

### Waitlist Association

**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

**Rationale**

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

- but, assuming that the presence of simultaneous processes is possible, this implementation shows a **race condition** \( \sim \) **lost wakeup**
  - while going to sleep, i.e. being “sleepy”, the process gets delayed
  - but in good faith of a sleeper, the “sleepy” process may be missed

---

### Indivisible Operation II

**Scope**

- \( P \) 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 \( P / V \)

**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
  - **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

---

#### 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

---

\(^5\) abbr. for first- or second-level interrupt handling, resp.
Semaphore Data Type

```c
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 respects:
  - 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:

```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...

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
In contrast to the shown general semaphore, a roused process has to recheck his waiting condition in P.

**General Process Management**

- **catch** has two variants, depending on the waitlist model (cf. p17):
  - 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 registered process**
- **overtakes the current process, reschedules the processor, and either performs a context switch or runs through the idle loop**
- **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**
- **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.

---

**Special Process Management**

- **Prevent Lost Wakeup**

```c
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**

- 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**

- **general semaphore**
  - #define when if
    - susceptible to erroneous rouse
  - overtaking impossible
    - gate <= 0 when a process aroused
    - rival process in P causes gate < 0
    - will be forced to sleep
    - aroused process may proceed
- **binary semaphore**
  - #define when while
    - gate = 1 when a process aroused
    - rival process in P causes gate = 0
    - is allowed to continue
    - aroused process has to wait

---

**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.*
Process States and State Transitions

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

- scheduler
- cause
- yield
- seize
- coast

Semaphore Gatekeeper

- as there is no single solution to protect P and V adequately, the wand attribute symbolises intention to application orientation
- if effective signalling (∨), i.e., waiting process
- depending on the mode of operation or use case, the wand acts differently
- doze (P¬), effective signalling (∨)
- assuming that processing elements are multiplexed [7, p.5], then:

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

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

Kernel-level Safeguard:
1. disable IRQ
2. lock critical section

Semaphore Gatekeeper (Ger.) Schrankenwärter, Türhüter

as there is no single solution to protect P and V adequately, the wand...
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.
Manuskript. – (dt.) Über Signalmasten

[2] Dijkstra, E. W.:
Cooperating Sequential Processes / Technische Universiteit Eindhoven.

[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

---

Reference List II

[5] Schröder-Preikschat, W. :
Concurrency.
In: [6], Kapitel 2

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
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

**principle pattern of a scheduler function to block a process**

- called by *coast* (cf. p. 27) and other functions to pause computation

```c
void block() {
    process_t *next, *self = being(OONESELF);
    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 */
}
```

- **choose next process to be dispatched to the processor**
- **ready list is empty, so the running process fades to the idle process**
- **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
  - ii. the running process, sent to sleep due to $P$, was roused due to $V$ (p. 27)