Agenda

Preface

Fundamentals
   Classification
   Characteristics

Implementation
   Data Structures
   Functions
   Mutex

Summary

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/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
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. 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.
    - decreases¹ 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.
    - increases¹ 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.

¹This does not only mean subtraction or addition, resp., in arithmetical terms.
multilateral synchronisation [5, p.15] of interacting processes
- the critical section is considered 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};
for (;;) {
    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 $V$.

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
- 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 availability control
- 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

```c
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” $P$)

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

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
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.

```c
semaphore_t data = {0}, store = {N}; /* N > 0 */
void producer() {
  for (;;) {
    P(&store);
    /* store acquired */
    P(&data);
    /* data acquired */
    V(&data);
    /* data released */
    V(&store);
  }
}
```

Semaphore v. Mutex II Conceptual Level

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:
  - i a category of methods for ensuring mutual exclusion or
  - ii the implementation of one of these methods in terms of an object

Semaphore v. Mutex I Technical Level

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).

- 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 ≤ 1 ⇒ 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...


- 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
    - 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])
    - 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

3If 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.
while going to sleep, i.e. being “sleepy”, the process gets delayed to block, a
inhibit FLIH
fall back on the elementary operations of the ISA level

if
atomic * sema = {
    void prolaag(s semaphore_t *sema) {
        atomic *sema = {
            if (ialeg(ssema))  /* at the moment, unavailable */
                sleep(&sema->wand);
        }
    }
    void verhoog(s semaphore_t *sema) {
        if (ubin(ssema))  /* 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 ~ 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

waitlist entry
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 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(s semaphore_t *sema) {
    atomic *sema = {
        if (!claim(ssema))
            sleep(&sema->wand);
    }
}

void verhoog(s semaphore_t *sema) {
    atomic *sema = {
        if (uban(ssema))
            rouse(&sema->wand);
    }
}

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:

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

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

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

Shallows Implementation

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

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

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

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

P and V Safeguarded Mutual Exclusion

```c
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 */
}
```

```c
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.
**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 */
}
```

**General Process Management**

Event Handling

**catch**
- exists in two variants, depending on the waitlist model (cf. p17):
  - i) store of a blocked-on mark in the process descriptor
  - 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.

**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:

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

```c
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

---

**Process States and State Transitions**

cf. [9, p.27]

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

- scheduler
- if effective signalling \(V\), i.e., waiting process
  - doze \(P\ →\), effective signalling \(← V\)
  - deep sleep \(P\), no overlapping \(V\)
Extents of Critical Section

Binary Semaphore devoid of Waitlist

- Let the sequence of instructions within P be as follows:
  1. Point at semaphore
  2. Address is blocked-on mark
  3. Check binary semaphore S_b
  4. Occupied, S_b already taken
  5. Deregulate P protection
  6. Fall asleep, dream about V

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:
    1. A binary semaphore used to actually protect the critical section and
    2. 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
  - A corresponding data type may be laid out as follows:

Outline

- Preface
- Fundamentals
- Classification
- Characteristics
- Implementation
- Data Structures
- Functions
- Mutex
- Summary

Outline

- Preface
- Fundamentals
- Classification
- Characteristics
- Implementation
- Data Structures
- Functions
- Mutex
- Summary

Copyright © wosch CS (WS 2014, LEC 7) Summary 32–37
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):
  - 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**

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.

**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 Universität Eindhoven.

2. Dijkstra, E. W.:
   Coordinating Sequential Processes / Technische Universität 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

4. Parnas, D. L.:
   Some Hypothesis About the “Uses” Hierarchy for Operating Systems / TH Darmstadt, Fachbereich Informatik.
   1976 (BSI 76/1). – Forschungsbericht

Semaphore v. Mutex III cf. p.14/15

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 $M$

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
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(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:
  - the idle/running process found itself ready-to-run on the ready list or
  - the running process, sent to sleep due to P, was roused due to V (p. 25)