Concurrent Systems

VII. Semaphore

Wolfgang Schröder-Preikschat

December 4, 2018

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/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”.*

- jumping-off point for *sleeping lock* (Ger. Schlafsperrre, [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.*

- 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 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 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

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 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

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

```c
semaphore_t data = {0};
void producer() {
  for (;;) {
    v(&data);
  }
}
void consumer() {
  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 beactable independent of P
- in order to complete, V depends on P
- 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 (;;) {
    P(&store);
    /* store released */
  }
}
```

- default value is N ≥ 0
- P must block out only if there is no store
- V indicates more store
- calling sequence
- V must beactable independent of P
- in order to complete, V depends on P
- 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 ≤ #P, must be completed by the same process
If no further process is ready to run, the current process but will not become “ready to run” or allocated the processor.

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

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

/* data released */
14
13
12
11
10
9
8
7
6
5
4
3
2
1

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.

if so, both may also entail context switching

if so, both may also entail context switching—“may” because:
- 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

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.

3\(^3\)see also p. 38
## 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.
    - 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

### Rationale

- **Logical**: To block, a **blocking** process descriptor joins a **queue data structure**.
  - To unblock, a process descriptor is removed from that queue data 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**.

### Implementation

- A characteristic by means of which **priority violation** will be prevented.

### Scope

- \( P \) and \( V \) itself constitute a **critical section**, likewise, that must be protected in order to function correctly.

### 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.

### 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. 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: **over-taking** of an aroused process.

- Upon return from \texttt{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 not least, concurrency had to be constricted to no more than what is absolutely necessary: reflect on \texttt{claim/sleep} and \texttt{unban/rouse}.

---

\(^5\) 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 respect
  - protect P and V against simultaneous processes
  - 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...

**P and V Safeguarded**

```c
void prolaag(semaphore_t *sema) {
    enter(&sema->wand); /* avert overlapped P or V */
    lodge(sema); /* raise claim to proceed */
    when (!avail(sema)) /* check for process delay */
        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
        leave(&sema->wand); /* allow P or V */
}
```

- exercise caution in the analysis of these program statements:
  - if applicable, “when” takes care of overtaking processes
  - if applicable, search for sleepers happens unconditionally
  - in case of (i) logical waitlist and (ii) strict binary semaphore

**Load/Store-Based Implementation for a Binary Semaphore:**

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

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

```c
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 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:

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

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

```c
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...

---

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 */
}
```

```c
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

```c
#define lodge(sema) sema->gate--
```

```c
#define avail(sema) sema->gate >= 0
```

```c
#define unban(sema) (sema->gate++ < 0)
```

- has two variants, depending on the waitlist model (cf. p17):
  - variant (i) writes to an own data structure of the current process, while variant (ii) manipulates a shared data structure
  - signals upcoming blocking (dormancy) of registered process

- 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

- #define when if
  - susceptible 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._
Process States and State Transitions

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

Semaphore Gatekeeper

- 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

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:
    - A binary semaphore used to actually protect the critical section and
    - 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 panics
  - A corresponding data type may be laid out as follows:
    ```c
    typedef volatile struct mutex {
      semaphore_t sema; /* binary semaphore */
      process_t *link; /* owning process or 0 */
    } mutex_t;
    ```

Acquire and Release Mutex

- Externally, a simple data type may be defined:
  ```c
  extern void panic(char*) __attribute__ (( noreturn ));
  ```
- A mutex is implemented as a semaphore:
  ```c
  void acquire(mutex_t *mutex) {
    P(&mutex->sema); /* lockout */
    mutex->link = being(ONESELF); /* register owner */
  }
  ```
  ```c
  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...
**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 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

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
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(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 */
}
```

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)