INF4140 - Models of concurrency

Fall 2016

September 9, 2016

Abstract

This is the "handout" version of the slides for the lecture (i.e., it's a rendering of the content of the slides in a way that does not waste so much paper when printing out). The material is found in [Andrews, 2000]. Being a handout-version of the slides, some figures and graph overlays may not be rendered in full detail, I remove most of the overlays, especially the long ones, because they don't make sense much on a handout/paper. Scroll through the real slides instead, if one needs the overlays.

This handout version also contains more remarks and footnotes, which would clutter the slides, and which typically contains remarks and elaborations, which may be given orally in the lecture. Not included currently here is the material about weak memory models.

1 Semaphores

7 September, 2015

1.1 Semaphore as sync. construct

Overview

- Last lecture: Locks and barriers (complex techniques)
  - No clear separation between variables for synchronization and variables to compute results
  - Busy waiting

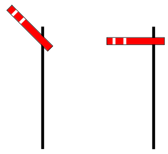
- This lecture: Semaphores (synchronization tool)
  - Used easily for mutual exclusion and condition synchronization.
  - A way to implement signaling (and scheduling).
  - implementable in many ways.
  - available in programming language libraries and OS

Outline

- Semaphores: Syntax and semantics
- Synchronization examples:
  - Mutual exclusion (critical sections)
  - Barriers (signaling events)
  - Producers and consumers (split binary semaphores)
  - Bounded buffer: resource counting
  - Dining philosophers: mutual exclusion – deadlock
  - Readers and writers: (condition synchronization – passing the baton

Semaphores

- Introduced by Dijkstra in 1968
- "inspired" by railroad traffic synchronization
- railroad semaphore indicates whether the track ahead is clear or occupied by another train



Properties

- Semaphores in concurrent programs: work similarly
- Used to implement
  - mutex and
  - condition synchronization
- Included in most standard libraries for concurrent programming
- also: *system calls* in e.g., Linux kernel, similar in Windows etc.

Concept

- *Semaphore*: special kind of shared program variable (with built-in sync. power)
- value of a semaphore: a *non-negative* integer
- can *only* be manipulated by two atomic operations:[1]

  P and V

  - **P:** (Passeren) Wait for signal – want to pass
    * effect: wait until the value is greater than zero, and decrease the value by one
  - **V:** (Vrijgeven) Signal an event – release
    * effect: increase the value by one
- nowadays, for libraries or sys-calls: other names are preferred (up/down, wait/signal, ...)
- different "flavors" of semaphores (binary vs. counting)
- a mutex: often (basically) a synonym for binary semaphore

Syntax and semantics

- declaration of semaphores:
  - sem s; default initial value is zero
  - sem s := 1;
  - sem s[4] := ([4] 1);
- semantics[2] (via "implementation"):

  P-operation P(s)

  $\langle \texttt{await}(s > 0)\ s := s - 1 \rangle$

  V-operation V(s)

  $\langle s := s + 1 \rangle$

[1] There are different stories about what Dijkstra actually wanted V and P to stand for.
[2] Semantics generally means "meaning"

Important: No direct access to the value of a semaphore.
E.g. a test like

```
if (s = 1) then ... else
```

is seriously *not* allowed!

Kinds of semaphores

- Kinds of semaphores

  General semaphore: possible values: all non-negative integers

  Binary semaphore: possible values: 0 and 1

  Fairness

  - as for await-statements.
  - In most languages: FIFO ("waiting queue"): processes delayed while executing P-operations are awaken in the order they where delayed

Example: Mutual exclusion (critical section)

Mutex[3] implemented by a binary semaphore

```
sem mutex := 1;
process CS[i = 1 to n] {
  while (true) {
  P(mutex);
    criticalsection;
  V(mutex);
  noncriticalsection;
}
```

Note:

- The semaphore is initially 1
- Always P before V → (used as) binary semaphore

Example: Barrier synchronization

Semaphores may be used for signaling events

```
sem arrive1 = 0, arrive2 = 0;
process Worker1 {
                    ...
        V(arrive1);         reach the barrier
        P(arrive2);         wait for other processes
                    ...
}
process Worker2 {
                    ...
        V(arrive2);         reach the barrier
        P(arrive1);         wait for other processes
                    ...
}
```

Note:

- signalling semaphores: usually initialized to 0 and
- signal with a V and then wait with a P

[3] As mentioned: "mutex" is also used to refer to a data-structure, basically the same as binary semaphore itself.

1.2 Producer/consumer

Split binary semaphores

Split binary semaphore
A set of semaphores, whose sum ≤ 1

mutex by split binary semaphores

- initialization: one of the semaphores =1, all others = 0
- discipline: all processes call P on a semaphore, before calling V on (another) semaphore

⇒ code between the P and the V

  - all semaphores = 0
  - code executed in mutex

Example: Producer/consumer with split binary semaphores

```
   T buf;  # one element buffer, some type T
sem empty := 1;
sem  full := 0;
```

```
process Producer {
  while (true) {
     P(empty);
     buff := data;
     V(full);
  }
}
```

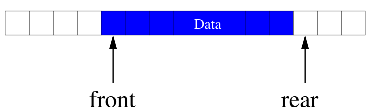
```
process Consumer {
  while (true) {
     P(full);
     data c := buff;
     V(empty);
  }
}
```

Note:

- remember also P/C with await + exercise 1
- `empty` and `full` are both binary semaphores, together they form a split binary semaphore.
- solution works with several producers/consumers

Increasing buffer capacity

- previously: tight coupling, the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- easy generalization: buffer of size n.
- loose coupling/asynchronous communcation ⇒ "buffering"
  - ring-buffer, typically represented
    * by an array
    * + two integers `rear` and `front`.
  - semaphores to keep track of the number of free/used slots ⇒general semaphore



Producer/consumer: increased buffer capacity

```
T buf[n]                      # array, elements of type T
int front := 0, rear := 0;    # ''pointers''
sem empty := n,
sem full := 0;
```

```
process Producer {
  while (true) {
      P(empty);
        buff[rear] := data;
        rear := (rear + 1) % n;
      V(full);
  }
}
```

```
process Consumer {
  while (true) {
      P(full);
        result := buff[front];
        front  := (front + 1) % n
      V(empty);
  }
}
```

several producers or consumers?

Increasing the number of processes

- several producers and consumers.
- New synchronization problems:
  - Avoid that two producers deposits to `buf[rear]` before `rear` is updated
  - Avoid that two consumers fetches from `buf[front]` before `front` is updated.
- Solution: additionally 2 binary semaphores for protection
  - `mutexDeposit` to deny two producers to deposit to the buffer at the same time.
  - `mutexFetch` to deny two consumers to fetch from the buffer at the same time.

Example: Producer/consumer with several processes

```
T buf[n]                              # array, elem's of type T
int front := 0, rear := 0;            # ''pointers''
sem empty := n,
sem full  := 0;
sem mutexDeposit, mutexFetch := 1; # protect the data stuct.
```

```
process Producer {
  while (true) {
    P(empty);
    P(mutexDeposit);
     buff[rear] := data;
     rear := (rear + 1) % n;
    V(mutexDeposit);
    V(full);
  }
}
```

```
process Consumer {
  while (true) {
    P(full);
    P(mutexFetch);
     result := buff[front];
     front  := (front + 1) % n
    V(mutexFetch);
    V(empty);
  }
}
```

1.3 Dining philosophers

Problem: Dining philosophers introduction

- famous sync. problem (Dijkstra)
- Five philosophers around a circular table.
- one fork placed between each pair of philosophers
- philosophers alternates between thinking and eating
- philosopher needs two forks to eat (and none for thinking)

Dining philosophers: sketch

```
process Philosopher [i = 0 to 4] {
  while true {
    think;
    acquire forks;
    eat;
    release forks;
  }
}
```

now: program the actions `acquire forks` and `release forks`

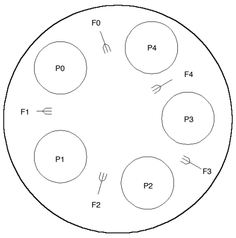
Dining philosophers: 1st attempt

- forks as semaphores
- philosophers: pick up left fork first

```
process Philosopher [i = 0 to 4] {
  while true {
    think;
    acquire forks;
    eat;
    release forks;
  }
}
```

```
sem fork[5]  := ([5] 1);
process Philosopher [i = 0 to 4] {
  while true {
    think;
    P(fork[i];
    P(fork[(i+1)%5]);
    eat;
    V(fork[i];
    V(fork[(i+1)%5]);
  }
}
```

[4]image from wikipedia.org



ok solution?

Example: Dining philosophers 2nd attempt

breaking the symmetry
To avoid deadlock, let 1 philospher (say 4) grab the right fork first

```
process Philosopher [i = 0 to 3] {
  while true {
    think;
    P(fork[i];
    P(fork[(i+1)%5]);
    eat;
    V(fork[i];
    V(fork[(i+1)%5]);
  }
}
```

```
process Philosopher4 {
  while true {
    think;
    P(fork[4];
    P(fork[0]);
    eat;
    V(fork[4];
    V(fork[0]);
  }
}
```

```
process Philosopher4 {
  while true {
    think;
    P(fork[0]);
    P(fork[4];
    eat;
    V(fork[4];
    V(fork[0]);
  }
}
```

Dining philosphers

- important illustration of problems with concurrency:
  - deadlock
  - but also other aspects: liveness and fairness etc.
- resource access
- connection to mutex/critical sections

1.4 Readers/writers

Example: Readers/Writers overview

- Classical synchronization problem
- Reader and writer processes, sharing access to a "database"
  - readers: read-only from the database

– writers: update (and read from) the database

- R/R access unproblematic, W/W or W/R: interference
  - writers need mutually exclusive access
  - When no writers have access, many readers may access the database

Readers/Writers approaches

- Dining philosophers: Pair of processes compete for access to "forks"
- Readers/writers: Different classes of processes competes for access to the database
  - Readers compete with writers
  - Writers compete both with readers and other writers

- General synchronization problem:
  - readers: must wait until no writers are active in DB
  - writers: must wait until no readers or writers are active in DB

- here: two different approaches
  1. Mutex: easy to implement, but "unfair"[5]
  2. Condition synchronization:
     - Using a split binary semaphore
     - Easy to adapt to different scheduling strategies

Readers/writers with mutex (1)

sem rw := 1

```
process Reader [i=1 to M] {
  while (true) {
     ...
    P(rw);

   read from DB

    V(rw);
  }
}
```

```
process Writer [i=1 to N] {
  while (true) {
     ...
    P(rw);

   write to DB

    V(rw);
  }
}
```

- safety ok
- but: unnessessarily cautious
- We want more than one reader simultaneously.

[5]The way the solution is "unfair" does not technically fit into the fairness categories we have introduced.

Readers/writers with mutex (2)

Initially:

```
int nr := 0;    # nunber of active readers
sem rw := 1     # lock for reader/writer mutex
```

```
process Reader [i=1 to M] {
  while (true) {
    ...
    < nr := nr + 1;
      if (nr=1) P(rw) > ;

    read from DB

    < nr := nr - 1;
      if (nr=0) V(rw) > ;
  }

}
```

```
process Writer [i=1 to N] {
  while (true) {
    ...

    P(rw);

    write to DB


    V(rw);
  }

}
```

Semaphore inside await statement? It's perhaps a bit strange, but works.

Readers/writers with mutex (3)

```
int      nr = 0;  # number of      active readers
sem      rw = 1;  # lock for reader/writer exclusion
sem mutexR = 1;   # mutex for readers

process Reader [i=1 to M] {
  while (true) {
    ...
      P(mutexR)
      nr := nr + 1;
      if (nr=1) P(rw);
      V(mutexR)

    read from DB

      P(mutexR)
     nr := nr - 1;
      if (nr=0) V(rw);
      V(mutexR)
  }
}
```

"Fairness"

What happens if we have a constant stream of readers? "Reader's preference"

Readers/writers with condition synchronization: overview

- previous mutex solution solved two separate synchronization problems
  - Readers and. writers for access to the database
  - Reader vs. reader for access to the counter
- Now: a solution based on **condition synchronization**

Invariant

reasonable invariant[6]

1. When a writer access the DB, no one else can
2. When no writers access the DB, one or more readers may

- introduce two counters:
  - `nr`: number of active readers
  - `nw`: number of active writers

The invariant may be:

RW: $(nr = 0 \text{ or } nw = 0) \text{ and } nw \leq 1$

Code for "counting" readers and writers

Reader:
```
< nr := nr + 1; >
read from DB
< nr := nr - 1; >
```

Writer:
```
< nw := nw + 1; >
write to DB
< nw := nw - 1; >
```

- maintain invariant ⇒ add sync-code
- decrease counters: not dangerous
- before increasing, check/synchronize:
  - before increasing `nr`: `nw = 0`
  - before increasing `nw`: `nr = 0 and nw = 0`

condition synchronization: without semaphores

Initially:

```
int nr := 0;   # nunber of active readers
int nw := 0;   # number of active writers
sem rw := 1    # lock for reader/writer mutex

## Invariant RW: (nr = 0 or nw = 0) and nw <= 1
```

```
process Reader [i=1 to M]{
  while (true) {
    ...
    < await (nw=0)
      nr := nr+1>;
    read from DB;
    < nr := nr − 1>
  }
}
```

```
process Writer [i=1 to N]{
  while (true) {
    ...
    < await (nr = 0 and nw = 0)
      nw := nw+1>;
    write to DB;
    < nw := nw − 1>
  }
}
```

[6] 2nd point: technically, not an invariant.

Condition synchr.: converting to split binary semaphores

implementation of `await`'s: possible via split binary semaphores

- May be used to implement different synchronization problems with different guards B_1, B_2...

General pattern

- entry[7] semaphore e, initialized to 1
- For each guard B_i:
  1. associate 1 counter and
  2. 1 delay-semaphore

  both initialized to 0
  * semaphore: delay the processes waiting for B_i
  * counter: count the number of processes waiting for B_i

⇒ for readers/writers problem: 3 semaphores and 2 counters:

```
sem e = 1;
sem r = 0; int dr = 0;      # condition reader:  nw == 0
sem w = 0; int dw = 0;      # condition writer:  nr == 0 and nw == 0
```

Condition synchr.: converting to split binary semaphores (2)

- `e, r` and `w` form a split binary semaphore.
- All execution paths start with a P-operation and end with a V-operation → Mutex

Signaling

We need a signal mechanism `SIGNAL` to pick which semaphore to signal.

- `SIGNAL`: make sure the invariant holds
- B_i holds when a process enters CR because either:
  - the process checks itself,
  - or the process is only signaled if B_i holds
- and another pitfall: Avoid deadlock by checking the counters before the delay semaphores are signaled.
  - `r` is not signalled (`V(r)`) unless there is a delayed reader
  - `w` is not signalled (`V(w)`) unless there is a delayed writer

Condition synchr.: Reader

```
   int nr := 0, nw = 0;        # condition variables (as before)
   sem e := 1;                 # entry semaphore
   int dr := 0; sem r := 0;    # delay counter + sem for reader
   int dw := 0; sem w := 0;    # delay counter + sem for writer
  # invariant RW: (nr = 0 ∨ nw = 0 ) ∧ nw ≤ 1
```

```
process Reader [i=1 to M]{   # entry condition: nw = 0
  while (true) {
    ...
      P(e);
      if (nw > 0) { dr := dr + 1;   # < await (nw=0)
                    V(e);           #     nr:=nr+1 >
                    P(r)};
      nr:=nr+1; SIGNAL;

      read from DB;

      P(e); nr:=nr -1; SIGNAL;      # < nr:=nr-1 >
  }
}
```

[7] Entry to the administractive CS's, not entry to data-base access

With condition synchronization: Writer

```
process Writer [i=1 to N]{   # entry condition: nw = 0 and nr = 0
  while (true) {
      ...
        P(e);                         # < await (nr=0 ∧ nw=0)
        if (nr > 0 or nw > 0) {       #      nw:=nw+1 >
            dw := dw + 1;
            V(e);
            P(w) };
        nw:=nw+1; SIGNAL;

        write to DB;

        P(e);nw:=nw -1; SIGNAL        # < nw:=nw-1>
  }
}
```

With condition synchronization: Signalling

- SIGNAL

```
if (nw = 0 and dr > 0) {
       dr := dr -1; V(r);                     # awake reader
  }
elseif (nr = 0 and nw = 0 and dw > 0) {
       dw := dw -1; V(w);                     # awake writer
  }
else
       V(e);                                  # release entry lock
```

References

[Andrews, 2000] Andrews, G. R. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.