Semaphores

INF4140

13.09.12

Lecture 3

---

[0] Book: Andrews - ch.04 (4.1 - 4.4)

Overview

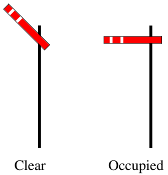
- Last lecture: Locks and Barriers (complex techniques)
  - No clear difference between variables for synchronization and variables for compute results.
  - Busy waiting.
- This lecture: Semaphores (synchronization tool)
  - Used easely for mutual exclusion and condition synchronization.
  - A way to implement signaling and (scheduling).
  - Can be implemented in many ways.

Outline

- Semaphores: Syntax and Semantics.
- Synchronization examples:
  - Mutual exclusion (Critical Section).
  - Barriers (signaling events).
  - Producers and consumers (split binary semaphores).
  - Bounded buffer (resource counting).
  - Dining philosophers (mutual exclusion - deadlock).
  - Reads and writers (condition synchronization - passing the baton).

Semaphores

- Introduced by Dijkstra in 1968
- Originates from railroad traffic synchronization
- Railroad semaphores indicates whether the track ahead is clear or occupied by another train

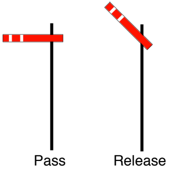


Properties

- Semaphores in concurrent programs work in a similar way
- Used to implement mutex and condition synchronization
- Included in most standard libraries for concurrent programming

Concept

- A semaphore is special kind of shared program variable
- The value of a semaphore is a non-negative integer
- Can *only* be manipulated by the following two atomic operations:
  - **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



Syntax and Semantics

- A semaphore is declared by either:
  - sem s; default initial value is zero
  - sem s $=$ 1;
  - sem s[4] $=$ ([4] 1);
- The **P** operation
  - syntax: P(s)
  - implementation (semantics): $<$ await (s $>$ 0) s $=$ s - 1; $>$
- The **V** operation
  - syntax: V(s)
  - implementation(semantics): $<$ s $=$ s $+$ 1; $>$

Important: No direct access to the value of a semaphore.
For instance a test like `if (s==1)` {...} *is 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 processes delayed while executing P-operations are awaken in the order they where delayed

Example: Mutual Exclusion (Critical Section)

Mutex implemented by a binary semaphore

```
sem mutex = 1;
process CS[i = 1 to n] {
    while (true) {
        P(mutex);
           critical section
        V(mutex);
           noncritical section
    }
}
```

Note:

- The semaphore is initially 1 and
- Always P before V → 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:

- Semaphores are usually initialized to 0 and
- Signal with a V and then wait with a P

Split binary semaphores

A set of semaphores may form a split binary semaphore if the sum of all semaphores never exceeds 1. Split binary semaphores can be used as follows to implement mutex.

- Consider a program with more than one binary semaphore
- Let one of these semaphores be initialized to 1 and the others to 0
- Let all processes call P on a semaphore, before calling V on (another) semaphore
- Then the code between the P and the V call will be executed in mutex. All semaphores will have the value 0.

Example: Producer/consumer with split binary semaphores

```
typeT buf; /* Buffer of some type T */
sem empty = 1, full = 0;

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

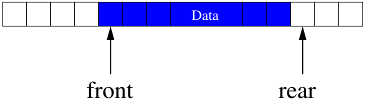
```
process Consumer{
      while (true){
         P(full);
         result = buf;
         V(empty);
           ...
      }
}
```

Note:

- `empty` and `full` are both binary semaphores, together they form a split binary semaphore.
- This solution works if there are several producers/consumers, but buffer capacity might be a problem

Increasing buffer capacity

- Previews example: the producer must wait for the consumer to empty the buffer before it can produce a new entry.
- It is a relatively simple task to generalize to a buffer of size n.
- We will use:
    - A ring-buffer represented by an array, and two integers `rear` and `front`.
    - General semaphores to keep track of the number of free slots



Example: Producer/consumer with increased buffer capacity

```
typeT buf[n]; /* Array av en type T */
int front = 0, rear = 0;
sem empty = n, full = 0;

 process Producer{                    process Consumer{
       while (true){                        while (true){
             ...                               P(full);
          P(empty);                            result = buf[front];
          buf[rear] = data;                    front = front + 1 % n;
          rear = rear + 1 % n;                 V(empty);
          V(full);                                ...
       }                                    }
 }                                    }
```

What if there are several producers or consumers?

Increasing the number of processes

- Let us consider a situation with 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.
- Solutions
    - Introduce a binary semaphore `mutexDeposit` to deny two producers to deposit to the buffer at the same time.
    - Introduce a binary semaphore `mutexFetch` to deny two consumers to fetch from the buffer at the same time.

Example: Producer/consumer with several processes

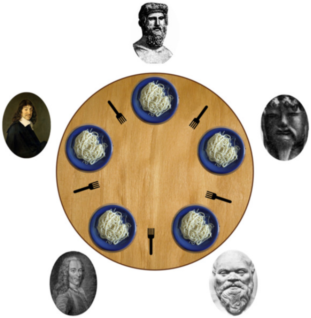
```
typeT buf[T]; int front = 0, rear = 0;
sem empty = n, full = 0, mutexDeposit = 1, mutexFetch = 1;
```

```
process Producer[i = 1 to M]{
        while (true){
                ...
            P(empty);
            P(mutexDeposit);
            buf[rear] = data;
            rear = rear + 1 % n;
            V(mutexDeposit);
            V(full);
        }
}
```

```
process Consumer[i = 1 to N]{
        while (true){
                ...
            P(full);
            P(mutexFetch);
            result = buf[front];
            front = front + 1 % n;
            V(mutexFetch);
            V(empty);
        }
}
```

Problem: Dining philosophers introduction

- Five philosophers sit around a circular table.
- One fork is placed between each pair of philosophers
- The philosophers alternates between thinking and eating
- A philosopher need two forks to eat.



---

[1] image from wikipedia.org

Example: Dining philosophers

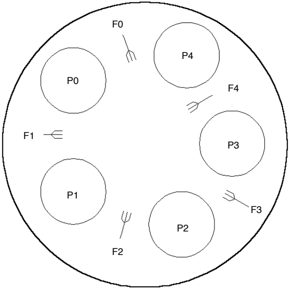
A sketch of the program may look like this:

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

We have to program the actions `acquire forks` and `release forks`

Example: Dining philosophers 1st attempt

- Let the forks be semaphores
- Let the philosophers pick up the left fork first



```
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]);
      }
}
```

Deadlock?

Example: Dining philosophers 2nd attempt

To avoid deadlock, let Philosopher4 grab the right fork first.

```
sem fork[5] = ( [5] 1)

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

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

Example: Readers/Writers overview

- Classical synchronization problem
- Reader and writer processes share a database

- Readers: reads from the database
- Writers: reads and updates the database

- Writers need mutually exclusive access
- When no writers have access, many readers may access the database

Example: Readers/Writers approaches

- Dining philosophers: Pair of processes competes 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
- We will look at two different approaches:
  - Mutex: easy to implement, but unfair
  - Condition Synchronization:
    - Using a split binary semaphore
    - Easy to adapt to different scheduling strategies

Example: 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);
      }
}
```

But: We want more than one reader simultaneously

Example: Readers/Writers with Mutex (2)

Give access to more than one reader

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

 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);
        }
}
```

We have to make the entry and exit of the CS atomic

Example: 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?

Example: Readers/Writers
with condition synchronization - overview

- The mutex solution solved two separate synchronization problems
  - Reader vs. writer for access to the database
  - Reader vs. reader for access to the counter
- We shall now look at a solution based on **condition synchronization**

Example: Readers/Writers with condition synchronization - invariant

- Let us try to find a reasonable invariant for the system.
- It must state that:
  - When a writer access the DB, no one else can
  - When no writers access the DB, one or more readers may
- Let us introduce two counters:
  - `nr` is the number of active readers
  - `nw` is the number of active writers
- The invariant may be:

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

Example: Readers/Writers
with condition synchronization - updating counters

We need code to update the counters

Reader:

```
< nr = nr + 1; >
Read from DB
< nr = nr - 1; >
```

Writer:

```
< nw = nw + 1; >
Write to DB
< nw = nw - 1; >
```

Put conditions in front of the atomic actions to preserve the invariant.

- Before increasing `nr`: `(nw == 0)`
- Before increasing `nw`: `(nr == 0 and nw == 0)`

Example: Readers/Writers
with condition synchronization - without semaphores

```
int nr = 0, nw = 0;
## 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; >
     }
}
```

Example: Readers/Writers with condition synchronization - converting to split binary semaphores

Implementation of `await`-statements may be done by split binary semaphores.

- May be used to implement different synchronization problems with different guards B_1, B_2...
- Use an entry-semaphore (e) initialized to 1
- For each guard B_i: associate one counter and one 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 the readers/writers problem, we need three semaphores and two 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
```

Example: Readers/Writers with condition synchronization - converting to split binary semaphores (2)

- `e`, `r` and `w` form a split binary semaphore.
- All execution paths starts with a P-operation and ends with a V-operation → Mutex
- We need a signal mechanism `SIGNAL` to pick which semaphore to signal.
- `SIGNAL` must make sure the invariant holds
- B_i holds when a process enters CR because either:
  - the process checks
  - the process is only signaled if B_i holds
- 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

Example: Readers/Writers with condition synchronization - with split binary semaphores

```
int nr = 0, nw = 0; int dr = 0; int dw = 0;
sem e = 1; sem r = 0; sem w = 0;
# RW: (nr == 0 ∨ nw == 0 ) ∧ nw <= 1

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

Example: Readers/Writers with condition synchronization - with split binary semaphores (2)

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

Example: Readers/Writers with condition synchronization - with split binary semaphores (3)

- SIGNAL

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