SPP (Synchro et Prog Parallèle)

Unit 4: Using locks:
Solving Some Typical Synchronisation problems with plain locks

François Taïani

Why look at typical problems?

- To recognise them when you see them
  - ➔ To be able to indentify and assess synchronisation needs
  - ➔ To be able to apply existing solutions
  - ➔ No need to reinvent the wheel when solutions exist
- To practice your understanding of locks
  - ➔ Locks can be tricky to use
- To organise your understanding of the field
  - ➔ How do problems and solutions relate to each other
  - ➔ A kind of mental map of what is out there

Two big families of problems

- When concurrent processes share resources
  - ➔ they can be in **competition** (contention)
  - ➔ or in **collaboration**
- Competition
  - ➔ amount of resources is limited
  - ➔ no all processes can use resources at the same time
  - ➔ synchronisation needed to decided who use what when
- Collaboration
  - ➔ processes work together to achieve a goal
  - ➔ their work is interdependent
  - ➔ synchronisation needed to control interdependencies

Examples

- Competition
  - ➔ 2 applications using the same printer
  - ➔ 2 users trying to book the same room at the same time
  - ➔ 2 players trying to pick a unique object in a game

- Collaboration
  - ➔ 1 application sending data to print to printer
  - ➔ 1 simulation providing results to another simulation
  - ➔ 1 player giving an object to another player in a game

Competition vs Collaboration

- Both often mixed in the same parallel program
  - ➔ see example with on-line game

- Solving 1 type of pb ➔ often need to solve other type
  - ➔ e.g. constructing a house: mainly collaboration
  - ➔ workers need to coordinate their work: workplan
  - ➔ but access to work plan is a competition problem

Typical problems

- What we have seen so far: **Mutual Exclusion**
  - ➔ a problem of competition
  - ➔ only one process / thread in the critical section at a time
  - ➔ we have seen one tool to solve it: locks (aka mutex)

- Today: Two typical problems of collaboration
  - ➔ readers / writers (simpler variant of getSum, transfer)
  - ➔ producer / consumer

Readers / Writers

- The context:
  - ➔ one variable
  - ➔ a number of threads write to it
  - ➔ a number of threads read from it (the latest value written)
  - ➔ e.g. a status variable (# pages printed, # files closed, etc.)
- Unsafe solution

```
shared int a

method read() is
  return a
end
```

```
method write(int x) is
  a = x
end
```

  - ➔ Quiz: How would you make it safe?

Readers / Writers

- Solutions
  - ➔ (1) use one plain lock to protect a: works but suboptimal
  - ➔ (2) use a read / write lock (aka shared lock, cf unit 2)

```
shared a // any resource, : image, string, etc
rw_lock l

method read() is
  l.lock_read()
  result = a // copy
  l.unlock()
  return result
end
```

```
method write(x) is
  l.lock_write()
  a = x
  l.unlock()
end
```

- Problem: What do you do if you don't have RW locks?

Readers / Writers

- Readers / writers problem reformulated:
  - ➔ one shared variable, with read and write operations
  - ➔ readers should block writers, but not other readers
  - ➔ writers should block both other writers and readers
  - ➔ we only have plain locks to solve the problem
- Start of a solution
  - ➔ intuition: readers should act “collectively”
  - ➔ keep count of # readers already accessing variable
  - ➔ 0 readers: variable accessible in writing and reading
  - ➔ ≥1 readers: variable only accessible in reading

RW Pb: First Attempt

```
shared a
shared int nb_readers = 0
lock l

method read() is
  if nb_readers==0 {
    l.lock()
  }
  nb_readers++
  result = a
  nb_readers--
  if nb_readers==0 {
    l.unlock()
  }
  return result
end
```

```
method write(x) is
  l.write()
  a = x
  l.unlock()
end
```

- This code is unfortunately unsafe. Why?

2nd attempt

```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
  if nb_readers==0 {
    l.lock()
  }
  count_l.lock()
  nb_readers++
  count_l.unlock()
  result = a
  count_l.lock()
  nb_readers--
  count_l.unlock()
  if nb_readers==0 {
    l.unlock()
  }
  return result
end
```

```
method write(x) is
  l.write()
  a = x
  l.unlock()
end
```

- This code is still faulty. Why?

Comment

- We need to protect the 2 `if nb_readers==0`
  - ➔ the “==0” is not guaranteed to be atomic
  - ➔ even if it is, test + increment/decrement need to be
  - ➔ otherwise can lead to unsafe execution

2nd attempt (again)

```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
  if nb_readers==0 {
    l.lock()
  }
  count_l.lock()
  nb_readers++
  count_l.unlock()
  result = a
  count_l.lock()
  nb_readers--
  count_l.unlock()
  if nb_readers==0 {
    l.unlock()
  }
  return result
end
```

```
method write(x) is
  l.lock()
  a = x
  l.unlock()
end
```

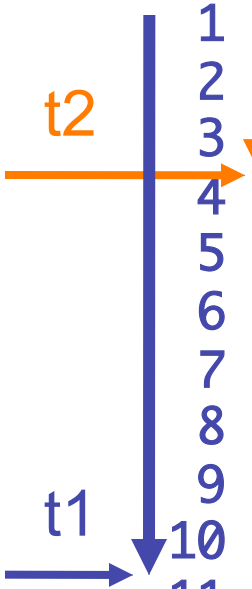
- Exercise
  - ➔ find an example of unsafe execution even if ==0 atomic

Example of pb

```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
1     if nb_readers==0 {
2       l.lock()
3     }
4     count_l.lock()
5     nb_readers++
6     count_l.unlock()
7     result = a
8     count_l.lock()
9     nb_readers--
10    count_l.unlock()
11    if nb_readers==0 {
12      l.unlock()
13    }
14    return result
end
```

- 3 threads t1, t2



t1:1,2,3,4,5,6,7
(l=locked,nb_readers==1)

t2:1,3
(l=locked,nb_readers==1)

t1:8,9,10,11,12,13,14)
(l=free,nb_readers==0)

t2:4,5,6
(l=free,nb_readers==1)

t2 about to access variable a but l is unlocked!!

Corrected Attempt: Solution 1

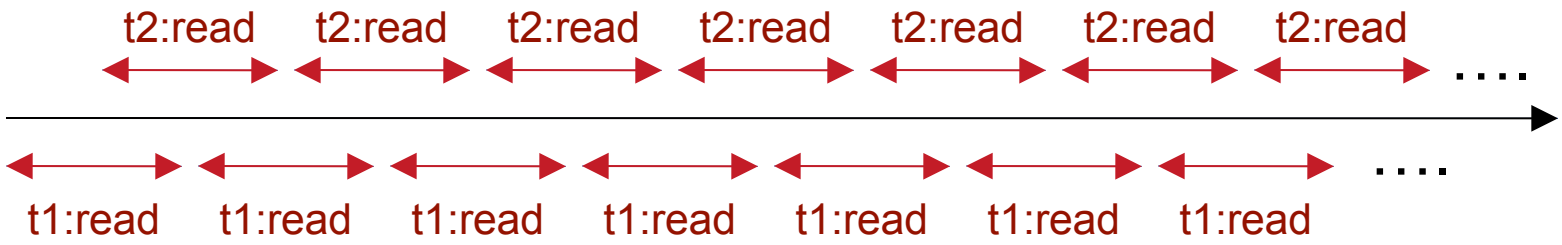
```
shared int a
shared int nb_readers = 0
lock l, count_l

method read() is
   count_l.lock()
   if nb_readers==0 {          // atomic
     l.lock()
   }
   nb_readers++                // atomic
   count_l.unlock()
   result = a
   count_l.lock()
   nb_readers--                // atomic
   if nb_readers==0 {          // atomic
     l.unlock()
   }
   count_l.unlock()
   return result
end

method write(x) is
   l.lock()
   a = x
   l.unlock()
end
```

Improving Solution 1

- Consider two threads doing reads:



  - ➔ What happens if a 3rd thread tries to do a write op?

Improving Solution 1: Fairness

- Solution 1 can lead to the starvation of writers
  - ➔ if reads are continuous and overlap
    then nb_readers always > 0
    and lock l never released
  - ➔ it's an unfair read / write lock (see Unit 2)
- How to solve this?
  - ➔ hypothesis: assume underlying locks are fair
  - ➔ intuition: get writer to prevent readers to continue
    i.e. get writer to get into the "queue" of readers
- How would you do this with 4 extra lines?

Fair and Safe: Solution 2

```
shared int a
shared int nb_readers = 0
lock l, count_l, fair_access

method read() is
    fair_access.lock()
    count_l.lock()
    if nb_readers==0 {
      l.lock() }
    nb_readers++
    count_l.unlock()
    fair_access.unlock()
    result = a
    count_l.lock()
    nb_readers--
    if nb_readers==0 {
      l.unlock() }
    count_l.unlock()
    return result
end
```

FIFO

FIFO

FIFO

corrected

```
method write(x) is
    fair_access.lock()
    l.lock()
    a = x
    l.unlock()
    fair_access.unlock()
end
```

FIFO

- variable access in order in which `fair_access` taken

Comments

- We have a solution to the readers/writers problem
- This also gives an implementation of read/write locks
  - ➔ note the two unlock operations (as in Java btw)
  - ➔ merging 2 unlock operations requires a bit of leg work (usually not worth the extra complexity)

Producers / Consumers

- 2nd typical collaboration synchronisation problem
  - ➔ one or more processes produce data
  - ➔ another or more processes consume this data
  - ➔ they communicate through a shared queue
- Different from previous example
  - ➔ no data should get lost
  - ➔ if the queue is empty, consumers should block
  - ➔ if the queue is full, producers should block
- We have already seen such a situation
  - ➔ scheduling queues in the blocking locks implementation

Producers / Consumers

- Assumptions: We have a bounded queue object with
  - add: add to the end of the queue
  - get: removed from the beginning of the queue (FIFO)
  - empty?: true if queue is empty
  - full?: true if queue is full
  - size: nb of elements currently in queue
- The queue is not thread-safe
  - concurrent operations can yield arbitrary results

Prod/Cons: 1st Attempt

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  result = q.get()
  l.unlock()
  return result
end
```

- What is the problem with this?

Prod/Cons: 1st Attempt

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  result = q.get()
  l.unlock()
  return result
end
```

- Empty and full cases not taken into account
  - ➔ empty: consumers should wait until item available
  - ➔ full: producers should wait until free slot available

Prod/Cons: 2nd Attempt

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  while (q.full?) {
  }
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  while (q.empty?) {
  }
  result = q.get()
  l.unlock()
  return result
end
```

- We now test if q is full (resp. empty)
  - ➔ But this is incorrect. Why? How would you correct it?

3rd Attempt: Solution 1

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  while (q.full?) {
    l.unlock()
    l.lock()
  }
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  while (q.empty?) {
    l.unlock()
    l.lock()
  }
  result = q.get()
  l.unlock()
  return result
end
```

- This now works correctly. (Safe and lively)
  - ➔ However not optimal? Find 2 reasons why.

Solution 1b with yield

- Problems with solution 1
  - 1. busy waiting: use yield (1b) to mitigate (but not eliminate!)
  - 2. does not allow concurrent produce / consume

```
shared queue q // q is not thead safe
lock l

method produce(x) is
  l.lock()
  while (q.full?) {
    l.unlock()
    yield()
    l.lock()
  }
  q.add(x)
  l.unlock()
end
```

```
method consume() is
  l.lock()
  while (q.empty?) {
    l.unlock()
    yield()
    l.lock()
  }
  result = q.get()
  l.unlock()
  return result
end
```

Summary

- Two general types of synchronisation problems
  - ➔ competition (mutual exclusion)
  - ➔ collaboration (readers and writers, consumer/producers)

- Lock-based solutions of
  - ➔ readers and writers (explain how RW locks are done)
  - ➔ consumer/producers (aka bounded buffer)

- Note on consumer/producers
  - ➔ uses busy waiting
  - ➔ we will see a better solution with semaphores in unit 5