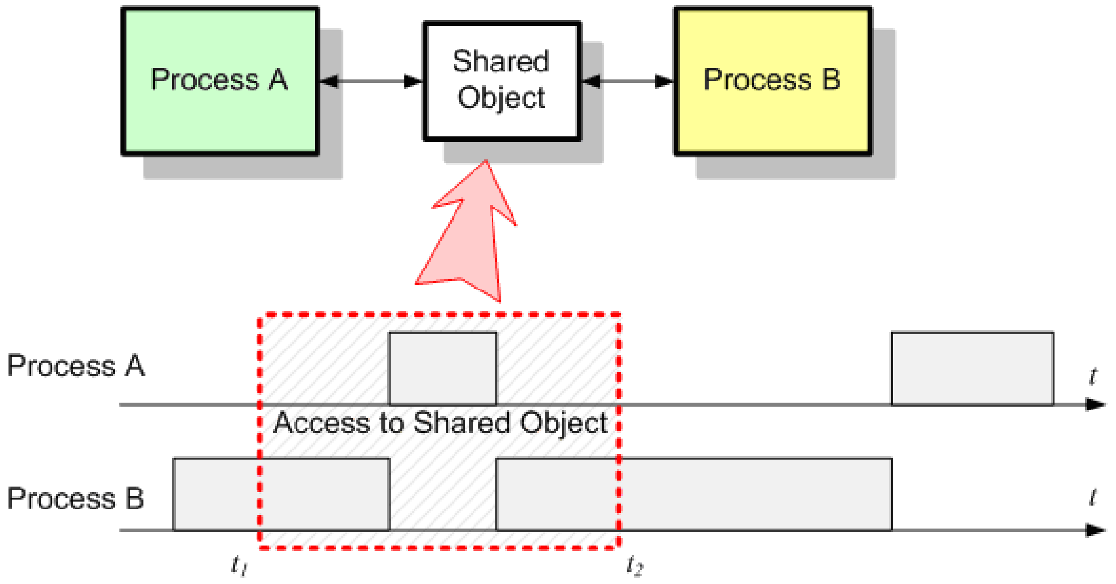
1

The Mutual Exclusion Problem

8.03.2018
Deepak Pal

The Mutual Exclusion Problem



In the following picture two independent processes – A and B – compete for an object that can be accessed by both between the moments $t_1$and t_2.

THE MUTUAL EXCLUSION PROBLEM

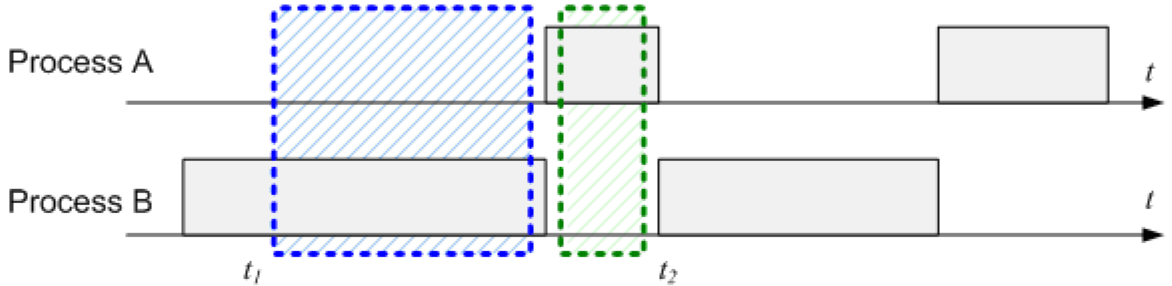
- The conflict comes from the fact that accessing the object requires a finite amount of time that is not 0 and the moments of preemption are completely unpredictable.

- Some may argue that most of the time the two processes will not access the resource in the same time and the probability for conflict is almost 0. Well, "***most of the time***" and "***is almost 0***" aren't good enough.

- **This has to change in "*all of the time*" and "*is always 0*" in order to have a reliable and determinist system.**

The Mutual Exclusion Problem

- The most logical way to solve this problem is to make sure that **only one process can access the shared object at one time**, and this access has to be complete or *atomic* (indivisible).
- The section of code we need to protect from concurrent access is known as *critical section*.



The result is obvious: the two processes will access the object sequentially and the process A will be delayed slightly because of process B keeping the CPU locked while the critical section is executed.

How to do the executing things not to trip over each other ?

- Eliminating undesirable **interleavings** is called the mutual exclusion problem.
- We need to identify **critical sections** that only one thread at a time can enter.
- We need to devise a **pre-Condition** and a **post-Condition** to keep two or more threads from being in their critical sections at the same time.

```
while (true) {
    non_Critical_Section;
    pre-Condition;
    critical_Section;
    post-Condition;
}
```

Problem for N processes

- **N processes** are executing, in an infinite loop, a sequence of instructions, which can be divided into two subsequences: the **critical section** and the **non-critical section**. The program must satisfy the **mutual exclusion property**: instructions from the critical sections of two or more processes must not be interleaved.

- The solution is described by inserting into the loop additional instructions that are to be executed by a process wishing to enter and leave its critical section – the **pre-Condition** and **post-Condition**, respectively. These protocols may require additional variables.

- A process may halt in its **non-critical section**. It may not halt during execution of its **conditions** or **critical section**. If one process halts in its **non-critical section**, it must not interfere with the operation of other processes.

Problem for N processes

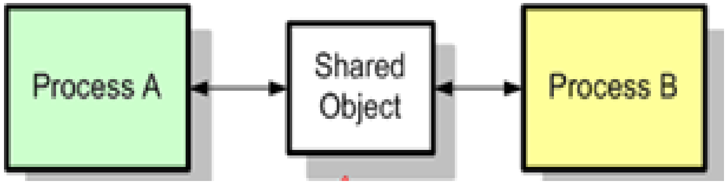
- The program must not **deadlock**. If some processes are trying to enter their critical sections then one of them must **eventually** succeed. The program is deadlocked if no process ever succeeds in making the transition from **pre-Condition** to critical section.

- There must be **no starvation** of any of the processes. If a process indicates its intention to enter its critical section by commencing execution of the **pre-Condition**, then eventually it must **succeed**.

- In the **absence of contention** for the critical section a single process wishing to enter its critical section will succeed. A good solution will have minimal overhead in this case.

The Mutual Exclusion Problem for 2 processes

- We will solve the mutual exclusion problem for two processes.



- One solution to the mutual exclusion problem for two processes is called **Dekker's algorithm**. We will develop this algorithm in step by-step sequence of incorrect algorithms: each will demonstrate some pathological behaviour that is typical of concurrent algorithms.

FIRST ATTEMPT

```
int turn=1;
```

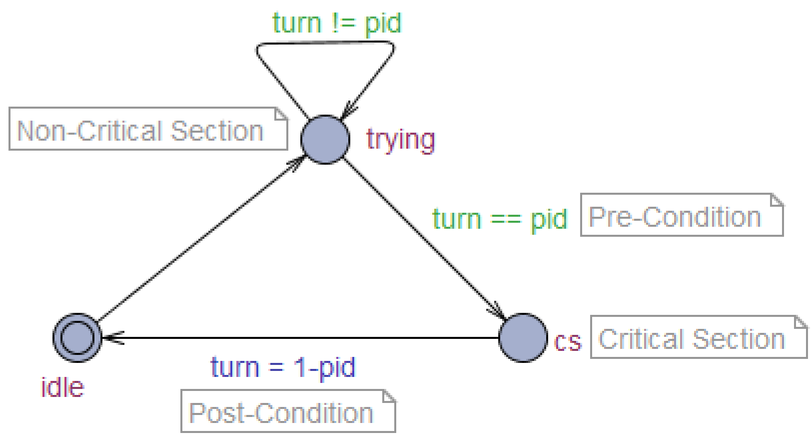
```
process P1
while (true) {
 nonCriticalSection1;
 while (turn == 2) {}
 criticalSection1;
 turn = 2;
}
end P1;
```

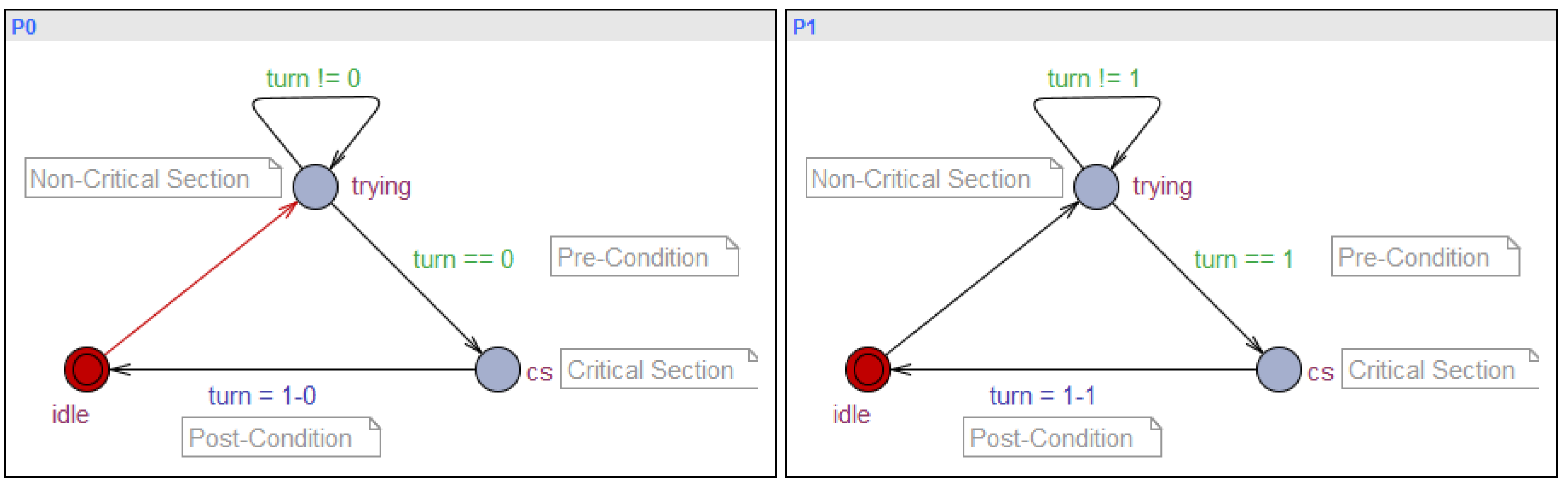
```
process P2
while (true) {
 nonCriticalSection2;
 while (turn == 1) {}
 criticalSection2;
 turn = 1;
}
end P2;
```

A single shared variable `turn` indicates whose turn it is to enter the critical section.

Mutual exclusion	No deadlock	No starvation	No starvation in absence of contention
✓	✓	✓	✗

UPPAAL MODEL





Second Attempt

```
int c1=1;
int c2=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 while (c2!=1) {}
 c1=0;
 criticalSection1;
 c1=1;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 while (c1!=1) {}
 c2=0;
 criticalSection2;
 c2=1;
}
end P2;
```

Each process Pi now has its own variable `ci`. Shared variable `ci==0` signals that Pi is about to enter its critical section.

Mutual exclusion ✘

Third Attempt

```
int c1=1;
int c2=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 c1=0;
 while (c2!=1) {}
 criticalSection1;
 c1=1;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 c2=0;
 while (c1!=1) {}
 criticalSection2;
 c2=1;
}
end P2;
```

In Attempt 2 the assignment `Ci=0` was effectively located in the critical section. Try moving it to the beginning of the pre-protocol, `Ci==0` now signals that Pi wishes to enter its critical section.

Mutual exclusion	No deadlock
✓	✗

FOURTH ATTEMPT

```
int c1=1;
int c2=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 c1=0;
 while (c2!=1) {
 c1=1;c1=0;
 }
 criticalSection1;
 c1=1;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 c2=0;
 while (c1!=1) {
 c2=1;c2=0;
 }
 criticalSection2;
 c2=1;
}
end P2;
```

The processes back off entering their critical sections if they detect both are trying to enter at the same time.

Mutual exclusion	No livelock	No starvation
✓	✗	✗

Dekker's Algorithm

```
int c1=1;
int c2=1;
int turn=1;
```

```
process P1
while (true) {
 nonCriticalSection1;
 c1=0;
 while (c2!=1)
   if (turn==2){
     c1=1;
     while (turn!=1) {}
     c1=0;
   }
 criticalSection1;
 c1=1; turn=2;
}
end P1;
```

```
process P2
while (true) {
 nonCriticalSection2;
 c2=0;
 while (c1!=1)
   if (turn==1){
     c2=1;
     while (turn!=2) {}
     c2=0;
   }
 criticalSection2;
 c2=1; turn=1;
}
end P2;
```

The threads now take turns at backing off.

Mutual exclusion	No deadlock	No starvation	No starvation in absence of contention
✓	✓	✓	✓

Assignments

- Assignment 1:
  - ATM System Model
    - (defend model behavior and property verification by 3-Mar-2018)

- Assignment 2:
  - Job Shop Model
    - (defend model behavior with deadlock by 3-Mar-2018)

- Assignment 3: Mutual Exclusion (discussion and problems in next lab)
  - Job Shop model (defend model without deadlock)
  - Uppaal models of all mutual exclusion algorithms (attempts)

Appendix: Proof For Attempts

First Attempt

Mutual exclusion	No deadlock	No starvation	No starvation in absence of contention
✓	✓	✓	✗

Mutual exclusion is satisfied

Proof: Suppose that at some point both processes are in their critical sections. Without loss of generality, assume that P1 entered at time t1, and that P2 entered at time t2, where t1 < t2. P1 remained in its critical section during the interval from t1 to t2.

At time t1, `turn==1`, and at time t2 `turn==2`. But during the interval t1 to t2 P1 remained in its critical section and did not execute its post-protocol which is the only means of assigning `2` to `turn`. At t2 `turn` must still be `1`, contradicting the previous statement.

PROOF: FIRST ATTEMPT

The solution cannot deadlock
Proof: For the program to deadlock each process must execute the test on `turn` infinitely often failing each time. Therefore, in P1 `turn==1` and in P2 `turn==2`, which is impossible.

There is no starvation
Proof: For starvation to exist one process must enter its critical section infinitely often, while the other executes its pre-protocol forever without progressing to its critical section.
But if P1 executes its even once, it will set `turn==2` in its post-protocol allowing P2 to enter its critical section.

There is starvation in the absence of contention
Proof: Suppose that P2 halts in its non-critical section: `turn` will never be changed from `2` to `1` . P1 may enter its critical section at most one more time. Once P1 sets `turn` to `2`, it will never again be able to progress from its pre-protocol to its critical section.

Proof: Second Attempt

Mutual exclusion
✘

Mutual exclusion is not satisfied

Proof: Consider the following interleaving beginning with the initial state.

1. P1 checks `c2` and finds `c2==1`.
2. P2 checks `c1` and finds `c1==1`.
3. P1 sets `c1` to `0`.
4. P2 sets `c2` to `0`.
5. P1 enters its critical section.
6. P2 enters its critical section.

Proof: Third Attempt

Mutual exclusion	No deadlock
✓	✗

Mutual exclusion is satisfied

Proof: Suppose that at some point both processes are in their critical sections. Without loss of generality, assume that P1 entered at time t1, and that P2 entered at time t2, where t1 < t2. P1 remained in its critical section during the interval from t1 to t2.

At time t1, `c1==0` and `c2==1` and at time t2 `c2==0` and `c1==1` . But during the interval t1 to t2 P1 remained in its critical section and did not execute its post-protocol which is the only means of assigning `1` to `c1`. At t2 `c1` must still be `0`, contradicting the previous statement.

PROOF: THIRD ATTEMPT

Mutual exclusion	No deadlock
✓	✗

The program can deadlock

Proof: Consider the following interleaving beginning with the initial state.

1. P1 sets `c1` to `0`.
2. P2 sets `c2` to `0`.
3. P1 tests `c2` and remains in the loop.
4. P2 tests `c1` and remains in the loop.

Both processes are locked forever in their pre-protocols.

PROOF: FOURTHATTEMPT

Mutual exclusion	No livelock	No starvation
✓	×	×

Livelock is a form of deadlock. In a deadlocked computation there is no possible execution sequence which succeeds. In a livelocked computation, there are successful computations, but there are one or more execution sequences in which no process enters its critical section.

Mutual exclusion is satisfied
Proof: Argument is the same as that for the third attempt.

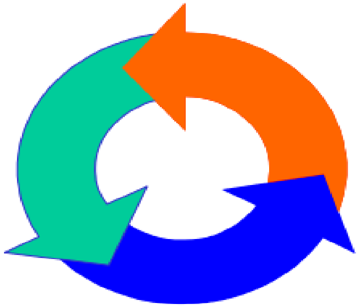
PROOF: FOURTH ATTEMPT

Mutual exclusion	No livelock	No starvation
✓	✗	✗

A process can be starved

Proof: Consider the following interleaving.

1. P1 sets c1 to 0.
2. P2 sets c2 to 0.
3. P2 checks c1 and resets c2 to 1.
4. P1 completes a full cycle:
   - checks c2
   - enters critical section
   - resets c1
   - enters non-critical section
   - sets c1 to 0
5. P2 sets c2 to 0.



P1 enters its critical section infinitely often, P2 remains indefinitely in its pre-protocol.

Proof: Fourth Attempt

Mutual exclusion	No livelock	No starvation
✓	✗	✗

A program can livelock

Proof: Consider the following interleaving.

1. P1 sets c1 to 0.
2. P2 sets c2 to 0.
3. P1 checks c2 and remains in the loop.
4. P2 checks c1 and remains in the loop.
5. P1 resets c1 to 1.
6. P2 resets c2 to 1.
7. P1 resets c1 to 0.
8. P2 resets c2 to 0.

As with deadlock both processes are locked in their pre-protocols. However, the slightest deviation from the above sequence will allow one process to enter its critical section.