Section 3

The Mutual Exclusion Problem
The Mutual Exclusion Problem

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-protocol and a post-protocol to keep two or more threads from being in their critical sections at the same time.

```java
while (true) {
    nonCriticalSection;
    preProtocol;
    criticalSection;
    postProtocol;
}
```
3.1: The Mutual Exclusion Problem for N processes

- N processes are executing, in an infinite loop, a sequence of instructions, which can be divided into two sub-sequences: 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-protocol and post-protocol, respectively. These protocols may require additional variables.

- A process may halt in its non-critical section. It may not halt during execution of its protocols or critical section. If one process halts in its non-critical section, it must not interfere with the operation of other processes.
The Mutual Exclusion 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-protocol 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-protocol, 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.
3.2: The Mutual Exclusion Problem for 2 processes

We will solve the mutual exclusion problem for two processes using Load and Store to common memory as the only atomic instructions.

```plaintext
C1=1;
if (c1==1) c2=c1;
```

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

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

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

Even if both processes are guaranteed not to halt this solution must be rejected. Why?
Second Attempt

Each process $P_i$ now has its own variable $C_i$. Shared variable $C_i == 0$ signals that $P_i$ is about to enter its critical section.
Second Attempt

Mutual exclusion is not satisfied

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

1. P1 checks $c_2$ and finds $c_2 == 1$.
2. P2 checks $c_1$ and finds $c_1 == 1$.
3. P1 sets $c_1$ to 0.
4. P2 sets $c_2$ to 0.
5. P1 enters its critical section.
6. P2 enters its critical section.
Third Attempt

In Attempt 2 the assignment $c_i=0$ was effectively located in the critical section. Try moving it to the beginning of the pre-protocol, $c_i==0$ now signals that $P_i$ wishes to enter its critical section.
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 \( t_1 \), and that P2 entered at time \( t_2 \), where \( t_1 < t_2 \). P1 remained in its critical section during the interval from \( t_1 \) to \( t_2 \).

At time \( t_1 \), \( c_1 = 0 \) and \( c_2 = 1 \) and at time \( t_2 \) \( c_2 = 0 \) and \( c_1 = 1 \). But during the interval \( t_1 \) to \( t_2 \) P1 remained in its critical section and did not execute its post-protocol which is the only means of assigning 1 to \( c_1 \). At \( t_2 \) \( c_1 \) must still be 0, contradicting the previous statement.
Third Attempt

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.
Fourth Attempt

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

A process can be starved
Proof: Consider the following interleaving.

1. P1 sets \( c_1 \) to 0.
2. P2 sets \( c_2 \) to 0.
3. P2 checks \( c_1 \) and resets \( c_2 \) to 1.
4. P1 completes a full cycle:
   - checks \( c_2 \)
   - enters critical section
   - resets \( c_1 \)
   - enters non-critical section
   - sets \( c_1 \) to 0
5. P2 sets \( c_2 \) to 0.

P1 enters its critical section infinitely often, P2 remains indefinitely in its pre-protocol.
A program can livelock
Proof: Consider the following interleaving.

1. P1 sets $c_1$ to 0.
2. P2 sets $c_2$ to 0.
3. P1 checks $c_2$ and remains in the loop.
4. P2 checks $c_1$ and remains in the loop.
5. P1 resets $c_1$ to 1.
6. P2 resets $c_2$ to 1.
7. P1 resets $c_1$ to 0.
8. P2 resets $c_2$ 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.
Dekker’s Algorithm

```java
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.
Dekker’s Algorithm

<table>
<thead>
<tr>
<th>Mutual exclusion</th>
<th>No deadlock</th>
<th>No starvation</th>
<th>No starvation in absence of contention</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
The difficulty in implementing mutual exclusion is caused by possible interleaving of load and store instructions. This difficulty disappears if a load and store are allowed in a single atomic instruction. This is provided on some computers as a test-and-set machine language instruction.

\text{testAndSet}(Li) \text{ is equivalent to the atomic execution of }

\begin{align*}
\text{Li} &= c; \\
c &= 1;
\end{align*}

where \( c \) is a global variable with initial value 0; and \( Li \) is a variable local to process \( Pi \).
Hardware Assisted Mutual Exclusion

int c=0;

process Pi
    private int Li = 0;
    while (true) {
        nonCriticalSectioni;
        do {
            testAndSet(Li);
        } while (Li != 0);
        criticalSectioni;
        c = 0;
    }
end Pi;

Li=c;
c=1;

Test_and_Set solution to the mutual exclusion problem for N processors
Hardware Assisted Mutual Exclusion

```
int c=0;

process Pi
    private int Li = 1;
    while (true) {
        nonCriticalSectioni;
        do {
            exchange(c,Li);
        } while ( Li != 0 );
        criticalSectioni;
        exchange(c,Li);
    }
end Pi;
```

Exchange solution to the mutual exclusion problem for N processors