Problem Description

Informally, a critical section is a code segment that accesses shared variables and has to be executed as an atomic action.

The critical section problem refers to the problem of how to ensure that at most one process is executing its critical section at a given time.

Important: Critical sections in different threads are not necessarily the same code segment!
Problem Description

Formally, the following requirements should be satisfied:

- **Mutual exclusion**: When a thread is executing in its critical section, no other threads can be executing in their critical sections.
- **Progress**: If no thread is executing in its critical section, and if there are some threads that wish to enter their critical sections, then one of these threads will get into the critical section.
- **Bounded waiting**: After a thread makes a request to enter its critical section, there is a bound on the number of times that other threads are allowed to enter their critical sections, before the request is granted.

In discussion of the critical section problem, we often assume that each thread is executing the following code.

It is also assumed that (1) after a thread enters a critical section, it will eventually exit the critical section; (2) a thread may terminate in the non-critical section.

```java
while (true) {
    entry section
    critical section
    exit section
    non-critical section
}
```
**Solution 1**

In this solution, `lock` is a global variable initialized to \textit{false}. A thread sets `lock` to \textit{true} to indicate that it is entering the critical section.

`boolean lock = false;`

```java
T0: while (true) {
    while (lock) { ; } (1)
    lock = true; (2)
    critical section (3)
    lock = false; (4)
    non-critical section (5)
}

T1: while (true) {
    while (lock) { ; } (1)
    lock = true; (2)
    critical section (3)
    lock = false; (4)
    non-critical section (5)
}
```

**Solution 1 is incorrect!**

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(1)</td>
<td>TO exits the while loop</td>
</tr>
<tr>
<td>context switch</td>
<td>context switch</td>
<td></td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
<td>TI exits the while loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock is set to true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TI enters the critical section</td>
</tr>
<tr>
<td>(3)</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>lock is set to true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TO enters the critical section</td>
</tr>
</tbody>
</table>
Solution 2

The threads use a global array \texttt{intendToEnter} to indicate their intention to enter the critical section.

\begin{verbatim}
boolean intendToEnter[] = {false, false};

T0: while (true) {
    while (intendToEnter[1]) { ; }
    (1)
    intendToEnter[0] = true;  (2)
    critical section  (3)
    intendToEnter[0] = false; (4)
    non-critical section  (5)
} 

T1: while (true) {
    while (intendToEnter[0]) { ; }
    (1)
    intendToEnter[1] = true;  (2)
    critical section  (3)
    intendToEnter[1] = false; (4)
    non-critical section  (5)
} 
\end{verbatim}

Solution 2 is incorrect!

\begin{verbatim}
T0
(1)
context switch
(2)
context switch
(3)
(4)
(5)

T1
(1)
context switch
(2)
context switch
(3)
(4)
(5)
\end{verbatim}

\begin{verbatim}
Comments
T0 exits the while loop
T0 exits the while loop
\texttt{intendToEnter[1]} is set to true
T1 enters the critical section
\texttt{intendToEnter[0]} is set to true
T0 enters the critical section
\end{verbatim}
Solution 3

The global variable \textit{turn} is used to indicate the next process to enter the critical section. The initial value of \textit{turn} can be 0 or 1.

\begin{verbatim}
int turn = 1;

T0: while (true) {
    while (turn != 0) { ; }
    (1)
    critical section
    (2)
    turn = 1;
    (3)
    non-critical section
    (4)
}

T1: while (true) {
    while (turn != 1) { ; }
    (1)
    critical section
    (2)
    turn = 0;
    (3)
    non-critical section
    (4)
}
\end{verbatim}

Solution 3 is incorrect!

\begin{tabular}{c|c|c}
\textbf{TO} & \textbf{T1} & \textbf{Comments} \\
\hline
(1) & (1) & T1 exits the while loop \\
(2) & (2) & T1 enters the critical section \\
(3) & (3) & \textit{turn} is set to 0 \\
(4) & (4) & T1 terminates in non-critical section \\
\textit{context switch} & & \\
(1) & (1) & T0 exits the while loop \\
(2) & (2) & T0 enters the critical section \\
(3) & (3) & \textit{turn} is set to 1 \\
(4) & (4) & T0 executes non-critical section \\
(1) & & T0 repeats (1) forever \\
\end{tabular}
Solution 4

When a thread finds that the other thread also intends to enter its critical section, it sets its own `intendToEnter` flag to false and waits until the other thread exits its critical section.

```java
boolean intendToEnter[] = {false, false};

T0:
while (true) {
    intendToEnter[0] = true; (1)
    while (intendToEnter[1]) { (2)
        intendToEnter[0] = false; (3)
        while (intendToEnter[1]) {} (4)
        intendToEnter[0] = true; (5)
    }
    critical section (6)
    intendToEnter[0] = false; (7)
    non-critical section (8)
}

T1:
while (true) {
    intendToEnter[1] = true; (1)
    while (intendToEnter[0]) { (2)
        intendToEnter[1] = false; (3)
        while (intendToEnter[0]) {} (4)
        intendToEnter[1] = true; (5)
    }
    critical section (6)
    intendToEnter[1] = false; (7)
    non-critical section (8)
}
```

Solution 4 is incorrect!

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
</tr>
<tr>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>context switch</td>
<td>context switch</td>
</tr>
<tr>
<td>context switch</td>
<td>context switch</td>
</tr>
<tr>
<td>(7)</td>
<td>(7)</td>
</tr>
<tr>
<td>(8)</td>
<td>(8)</td>
</tr>
<tr>
<td>(1)</td>
<td>(1)</td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
</tr>
<tr>
<td>(6)</td>
<td>(6)</td>
</tr>
<tr>
<td>context switch</td>
<td>context switch</td>
</tr>
<tr>
<td>context switch</td>
<td>context switch</td>
</tr>
<tr>
<td>(4)</td>
<td>(4)</td>
</tr>
<tr>
<td>repeat infinitely</td>
<td>repeat infinitely</td>
</tr>
</tbody>
</table>

**Comments**

- `intendToEnter[0]` is set to true
- T0 exits while loop
- T0 enters critical section; `intendToEnter[0]` is true
- T1 enters while (`intendToEnter[0]`) loop
- `intendToEnter[1]` is set to true
- T1 enters second while(`intendToEnter[0]`) loop
- `intendToEnter[0]` is set to false
- T0 executes the non-critical section
- `intendToEnter[0]` is set to true
- T0 exits while loop
- T0 enters critical section; `intendToEnter[0]` is true
- T1 is still waiting for `intendToEnter[0]` to be false
- `intendToEnter[0]` is set to false
- T0 executes the non-critical section
- `intendToEnter[0]` is set to true
- T0 exits while loop
- T0 enters critical section; `intendToEnter[0]` is true
- T1 is still waiting for `intendToEnter[0]` to be false
How to check a solution

Informally, we should consider three important cases:

1. One thread intends to enter its critical section, and the other thread is not in its critical section or in its entry section.

2. One thread intends to enter its critical section, and the other thread is in its critical section.

3. Both threads intend to enter their critical sections.

Peterson's algorithm

Peterson's algorithm is a combination of solutions (3) and (4).

```java
boolean intendToEnter[] = {false, false};
int turn; // no initial value for turn is needed.

T0: while (true) {
    intendToEnter[0] = true; (1)
    turn = 1; (2)
    while (intendToEnter[1] && turn == 1) { ; } (3)
    critical section (4)
    intendToEnter[0] = false; (5)
    non-critical section (6)
}

T1: while (true) {
    intendToEnter[1] = true; (1)
    turn = 0; (2)
    while (intendToEnter[0] && turn == 0) { ; } (3)
    critical section (4)
    intendToEnter[1] = false; (5)
    non-critical section (6)
}
```
**Peterson's algorithm**

Informally, we consider the following cases:

1. Assume that one thread, say T0, intends to enter its critical section and T1 is not in its critical section or its entry-section. Then intendToEnter[0] is true and intendToEnter[1] is false and T0 will enter the critical section immediately.

2. Assume that thread T0 intends to enter its critical section and T1 is in its critical section. Since turn = 1, T0 loops at statement (3). After the execution of (5) by T1, if T0 resumes execution before T1 intends to enter again, T0 enters its critical section immediately; otherwise, see case (3).
Peterson's algorithm

3. Assume both threads intend to enter the critical section, i.e. both threads have set their \textit{intendToEnter} flags to true. The thread that first executes "\textit{turn} = ...;" waits until the other thread executes "\textit{turn} = ...;" and then enters its critical section. The other thread will be the next thread to enter the critical section.
**Bakery algorithm**

Bakery algorithm is used to solve the n-process critical section problem. The main idea is the following:

- When a thread wants to enter a critical section, it gets a ticket. Each ticket has a number.
- Threads are allowed to enter the critical section in ascending order of their ticket numbers.

**Version 1**

```java
int number[n]; // array of ticket numbers, initially all elements of number is 0

while (true) {
    number[i] = max(number) + 1; (1)
    for (int j = 0; j < n; j++) { (2)
        while (j != i && number[j] != 0 && (number[j], j) < (number[i], i)) { ; } (3)
    }
    critical section (4)
    number[i] = 0; (5)
    non-critical section (6)
}
```

- \((a, b) < (c, d)\) if \(a < c\) or \(a = c\ and\ b < d\)
- \(\text{max}(\text{number})\) is the \(\text{max}\) value of all the elements in \(\text{number}\)
**Version 1 is incorrect!**

<table>
<thead>
<tr>
<th>T0</th>
<th>T1</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1)</td>
<td>(1)</td>
<td>T0 executes $\text{max}(\text{number}) + 1$, switch occur before 1 is assigned to number[0]</td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
<td>T1 sets number[1] to 1</td>
</tr>
<tr>
<td>(3)</td>
<td>(3)</td>
<td>T1 starts for loop</td>
</tr>
<tr>
<td>(4)</td>
<td>(4)</td>
<td>T1 exits while and for loop</td>
</tr>
<tr>
<td></td>
<td></td>
<td>T1 enters critical section</td>
</tr>
<tr>
<td>(1)</td>
<td>(1)</td>
<td>T0 assigns 1 (not 2) to number[0]</td>
</tr>
<tr>
<td>(2)</td>
<td>(2)</td>
<td>T0 starts for loop</td>
</tr>
<tr>
<td>(3)</td>
<td>(3)</td>
<td>T0 exits the while and for loops</td>
</tr>
<tr>
<td>(4)</td>
<td>(4)</td>
<td>T0 enters critical section</td>
</tr>
</tbody>
</table>

---

**Bakery algorithm**

```java
int number[n]; // array of ticket numbers, initially all elements of number is 0
boolean choosing[n]; // initially all elements of choosing is false
```

```java
while (true) {
  choosing[i] = true;               // (1)
  number[i] = max(number) + 1;      // (2)
  choosing[i] = false;              // (3)
  for (int j = 0; j < n; j++) {
    while (choosing[j]) {           // (4)
      while (j != i && number[j] != 0 &&
        (number[j], j) < (number[i], i)) { // (5)
        number[j] = 0;                 // (6)
      }
    }
  }

  critical section                  // (7)
  number[i] = 0;                    // (8)
  non-critical section              // (9)
}
```

- $(a, b) < (c, d)$ if $a < c$ or $(a = c$ and $b < d)$
- $\text{max}(\text{number})$ is the max value of all the elements in number
Bakery algorithm

Let us consider the following cases:

1. One thread, say Ti, intends to enter its critical section and no other thread is in its critical section or entry section. Then number[j] = 1 and number[j], where j ≠ i, is 0. Thus, Ti enters its critical section immediately.

2. One thread, say Ti, intends to enter its critical section and Tk, k ≠ i, is in its critical section. Then at (6), when j = k, number[j] < number[i]. Thus, Ti is delayed at (6) until Tk executes (8).

3. Two or more threads intend to enter their critical sections and no other threads is in its critical section. Assume that Tk and Tm, where k < m, intend to enter. Consider the possible relationships between number[k] and number[m]:
   - number[k] < number[m]. Tk enters its critical section since (number[m], m) > (number[k], k).
   - number[k] == number[m]. Tk enters its critical section since (number[m], m) > (number[k], k).
   - number[k] > number[m]. Tm enters its critical section since (number[k], k) > (number[m], m).