Concepts of Concurrent Computation

Sebastian Nanz

Lecture 3: Synchronization Algorithms
Today's lecture

In this lecture you will learn about:

• the mutual exclusion problem, a common framework for evaluating solutions to the problem of exclusive resource access
• solutions to the mutual exclusion problem (Peterson's algorithm, the Bakery algorithm) and their properties
• ways of proving properties for concurrent programs
The mutual exclusion problem
Mutual exclusion

• As discussed in the last lecture, race conditions can corrupt the result of a concurrent computation if processes are not properly synchronized
• We want to develop techniques for ensuring mutual exclusion
• *Mutual exclusion*: a form of synchronization to avoid the simultaneous use of a shared resource
• To identify the program parts that need attention, we introduce the notion of a critical section
• *Critical section*: part of a program that accesses a shared resource.
The mutual exclusion problem (1)

• We assume to have \( n \) processes of the following form:

```
while true loop
  entry protocol
  critical section
  exit protocol
  non-critical section
end
```

• Design the entry and exit protocols to ensure:
  • **Mutual exclusion**: At any time, at most one process may be in its critical section
  • **Freedom from deadlock**: If two or more processes are trying to enter their critical sections, one of them will eventually succeed
  • **Freedom from starvation**: If a process is trying to enter its critical section, it will eventually succeed
The mutual exclusion problem (2)

- Further important conditions:
  - Processes can communicate with each other only via atomic read and write operations
  - If a process enters its critical section, it will eventually exit from it
  - A process may loop forever or terminate while being in its non-critical section
  - The memory locations accessed by the protocols may not be accessed outside of them
• Synchronization mechanisms based on the ideas of entry- and exit-protocols are called *locks*
• They can typically be implemented as a pair of functions:

```
lock
  do
    entry protocol
  end
end

unlock
  do
    exit protocol
  end
```
Towards a solution

• The mutual exclusion problem is quite tricky: in the 1960's many incorrect solutions were published
• We will work along a series of failing attempts until establishing a solution
• We will start with trying to find a solution for only two processes
Busy waiting

• We will use the following statement in pseudo code

```pseudo
await b
```

which is equivalent to

```pseudo
while not b loop end
```

• This type of waiting is called *busy waiting* or "spinning"
• Busy waiting is inefficient on multitasking systems
• Busy waiting makes sense if waiting times are typically so short that a context switch would be more expensive
• Therefore spin locks (locks using busy waiting) are often used in operating system kernels
Solution attempt I

- **First idea**: use two variables `enter1` and `enter2`; if `enteri` is true, it means that process `P_i` intends to enter the critical section

<table>
<thead>
<tr>
<th>enter1 := false</th>
<th>enter2 := false</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>while true loop</td>
<td>while true loop</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>while true loop</td>
<td>await not enter1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>await not enter2</td>
<td>enter2 := true</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>enter1 := true</td>
<td>critical section</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>critical section</td>
<td>enter2 := false</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>enter1 := false</td>
<td>non-critical section end</td>
</tr>
<tr>
<td>6</td>
<td>end</td>
</tr>
</tbody>
</table>

non-critical section
Solution attempt I is incorrect

- The solution attempt fails to ensure mutual exclusion
- The two processes can end up in their critical sections at the same time, as demonstrated by the following execution sequence

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>1</td>
<td><code>await not enter1</code></td>
</tr>
<tr>
<td>P1</td>
<td>1</td>
<td><code>await not enter2</code></td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td><code>enter1 := true</code></td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td><code>enter2 := true</code></td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td><code>critical section</code></td>
</tr>
<tr>
<td>P1</td>
<td>3</td>
<td><code>critical section</code></td>
</tr>
</tbody>
</table>
Solution attempt II

- When analyzing the failure, we see that we set the variable `enter i` only after the `await` statement, which is guarding the critical section
- **Second idea:** switch these statements around

```
enter1 := false
enter2 := false

P1
while true loop
  1. enter1 := true
  2. await not enter2
  3. critical section
  4. enter1 := false
  5. non-critical section
end

P2
while true loop
  1. enter2 := true
  2. await not enter1
  3. critical section
  4. enter2 := false
  5. non-critical section
end
```
Solution attempt II is incorrect

• The solution provides mutual exclusion
• However, the processes can deadlock:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>1</td>
<td>enter1 := true</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>enter2 := true</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>await not enter1</td>
</tr>
<tr>
<td>P1</td>
<td>2</td>
<td>await not enter2</td>
</tr>
</tbody>
</table>
Solution attempt III

- **Third idea**: let's try something new, namely a single variable `turn` that has value `i` if it's $P_i$'s turn to enter the critical section.

<table>
<thead>
<tr>
<th>turn := 1 or turn := 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>P1</strong></td>
</tr>
<tr>
<td>1 while true loop</td>
</tr>
<tr>
<td>2 await turn = 1</td>
</tr>
<tr>
<td>3 critical section</td>
</tr>
<tr>
<td>4 turn := 2</td>
</tr>
<tr>
<td>5 non-critical section</td>
</tr>
<tr>
<td>end</td>
</tr>
</tbody>
</table>
Proving correctness of solution attempt III

- Solution attempt III looks good to us, let's try to prove it correct.
- Draw the related transition system; states are labeled with triples \((i, j, k)\): program pointer values \(P1 \triangleright i\) and \(P2 \triangleright j\), and value of the variable turn = \(k\).
Proving correctness of solution attempt III

• **Solution attempt III satisfies mutual exclusion**

  *Proof.* Mutual exclusion expressed as LTL formula:
  \[
  G \neg (P_1 > 2 \land P_2 > 2)
  \]
  Easy to see that this formula holds, as there are no states of the form (2, 2, k).

• **Solution attempt III is deadlock-free**

  *Proof.* Deadlock-freedom expressed as LTL formula:
  \[
  G ((P_1 > 1 \land P_2 > 1) \rightarrow F (P_1 > 2 \lor P_2 > 2))
  \]
  We have to examine the states (1, 1, 1) and (1, 1, 2); in both cases, one of the processes is enabled to enter its critical section.
Another setback

• Let's check starvation-freedom
• Expressed as LTL formula: for $i = 1, 2$
  \[ G (P_i \triangleright 1 \rightarrow F (P_i \triangleright 2)) \]
• Recall: processes may terminate in non-critical section
• A problematic case is (1, 4, 2): variable turn = 2, P1 trying to enter critical section (although not its turn), P2 in non-critical section
• If P2 terminates, turn will never be set to 1: P1 will starve
Peterson's algorithm
Peterson's algorithm

- Peterson's algorithm combines the ideas of solution attempts II and III
- If both processes have set their enter-flag to true, then the value of turn decides who may enter the critical section

```plaintext
enter1 := false
ten2 := false
turn := 1 or turn := 2
```

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>while true loop</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>enter2 := true</td>
</tr>
<tr>
<td>2</td>
<td>turn := 1</td>
</tr>
<tr>
<td>3</td>
<td>await not enter1 or turn = 2</td>
</tr>
<tr>
<td>4</td>
<td>critical section</td>
</tr>
<tr>
<td>5</td>
<td>enter2 := false</td>
</tr>
<tr>
<td>6</td>
<td>non-critical section</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
</tbody>
</table>
Peterson's algorithm: mutual exclusion

• *Peterson's algorithm satisfies mutual exclusion*

Proof.

• Assume that both P1 and P2 are in their critical section and that P1 entered before P2

• When P1 entered the critical section we have enter1 = true, and P2 must thus have seen turn = 2 upon entering its critical section

• P2 could not have executed line 2 after P1 entered, as this sets turn = 1 and would have excluded P2, as P1 does not change turn while being in the critical section

• However, P2 could not have executed line 2 before P1 entered either because then P1 would have seen enter2 = true and turn = 1, although P2 should have seen turn = 2

• Contradiction
Peterson's algorithm: starvation-freedom

• *Peterson's algorithm is starvation-free*

*Proof.*

• Assume P1 is forced to wait in the entry protocol forever
• P2 can eventually do only one of three actions:
  1. Be in its non-critical section: then enter2 is false, thus allowing P1 to enter.
  2. Wait forever in its entry protocol: impossible because turn cannot be both 1 and 2
  3. Repeatedly cycle through its code: then P2 will set turn to 1 at some point and never change it back
Peterson's algorithm for \( n \) processes

- Up until now, we have only seen a solution to the mutual exclusion problem for two processes; the problem is however posed for \( n \) processes
- Peterson's algorithm has a direct generalization

\[
\begin{array}{|c|c|c|}
\hline
\text{enter}[1] := 0; \ldots; \text{enter}[n] := 0 \\
\text{turn}[1] := 0; \ldots; \text{turn}[n - 1] := 0 \\
\hline
\text{P}_i \\
\hline
1 & \text{for } j = 1 \text{ to } n - 1 \text{ do} \\
2 & \quad \text{enter}[i] := j \\
3 & \quad \text{turn}[j] := i \\
4 & \quad \text{await (for all } k \neq i : \text{enter}[k] < j) \text{ or turn}[j] \neq i \\
5 & \text{end} \\
6 & \text{critical section} \\
7 & \text{enter}[i] := 0 \\
8 & \text{non-critical section} \\
\hline
\end{array}
\]
Peterson's algorithm for \( n \) processes

- Every process has to go through \( n - 1 \) stages to reach the critical section: variable \( j \) indicates the stage
- \texttt{enter}[i]: stage the process \( P_i \) is currently in
- \texttt{turn}[j]: which process entered stage \( j \) last
- Waiting: \( P_i \) waits if there are still processes at higher stages, or if there are processes at the same stage unless \( P_i \) is no longer the last process to have entered this stage
- Idea for mutual exclusion proof: at most \( n - j \) processes can have passed stage \( j \) => at most \( n - (n - 1) = 1 \) processes can be in the critical section
The Bakery algorithm
Fairness again

- Freedom from starvation still allows that processes may enter their critical sections before a certain, already waiting process is allowed access.
- We study an algorithm that has very strong fairness guarantees.
Bounded waiting

• The following definitions help analyze the fairness with respect to process waiting in mutual exclusion algorithms.

• **Bounded waiting**: If a process is trying to enter its critical section, then there is a bound on the number of times any other process can enter its critical section before the given process does so.

• **r-bounded waiting**: If a process tries to enter its critical section then it will be able to enter before any other process is able to enter its critical section \( r + 1 \) times.

• This means: bounded waiting = there exists an \( r \) such that the waiting is \( r \)-bounded.

• **First-come-first-served**: 0-bounded waiting.
Relating the definitions

- starvation-freedom $\Rightarrow$ deadlock-freedom
- starvation-freedom $\not\Rightarrow$ bounded waiting
- bounded waiting $\not\Rightarrow$ starvation-freedom
- bounded waiting + deadlock-freedom $\Rightarrow$ starvation-freedom

**deadlock-freedom**  If two or more processes are trying to enter their critical sections, one of them will eventually succeed.

**starvation-freedom**  If a process is trying to enter its critical section, it will eventually succeed.

**bounded waiting**  If a process is trying to enter its critical section, then there is a bound on the number of times any other process can enter its critical section before the given process does so.
Peterson's algorithm: no bounded waiting

- Assume a scenario with three competing processes

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>2</td>
<td>enter[1] := 1</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>enter[2] := 1</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>turn[1] := 2</td>
</tr>
<tr>
<td>P3</td>
<td>2</td>
<td>enter[3] := 1</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>turn[1] := 3</td>
</tr>
</tbody>
</table>

\[ \text{turn}[1] \neq 2: \text{P2 can proceed} \]

P2 \[\ldots\text{enters + leaves critical section}\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>P2</td>
<td>2</td>
<td>enter[2] := 1</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>turn[1] := 2</td>
</tr>
<tr>
<td>P3</td>
<td>3</td>
<td>turn[1] := 3</td>
</tr>
</tbody>
</table>

\[ \text{turn}[1] \neq 3: \text{P3 can proceed} \]

P3 \[\ldots\text{enters + leaves critical section}\]

\[ \ldots\text{P3 can unblock P2 etc.}\]

- P2 and P3 can overtake P1 unboundedly often
- Still P1 is not starved as it eventually (fairness) executes turn[1] := 1 and can proceed into the critical section
The bakery algorithm: first attempt

• **Idea:** ticket systems for customers, at any turn the customer with the lowest number will be served
• **number[i]:** ticket number drawn by a process $P_i$
• **Waiting:** until $P_i$ has the lowest number currently drawn

<table>
<thead>
<tr>
<th>number[1] := 0; ...; number[n] := 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_i$</td>
</tr>
<tr>
<td>1 number[i] := 1 + max(number[1], ..., number[n])</td>
</tr>
<tr>
<td>2 <strong>for all</strong> j != i <strong>do</strong></td>
</tr>
<tr>
<td>3 <strong>await</strong> number[j] = 0 <strong>or</strong> number[i] &lt; number[j]</td>
</tr>
<tr>
<td><strong>end</strong></td>
</tr>
<tr>
<td>4 <strong>critical section</strong></td>
</tr>
<tr>
<td>5 number[i] := 0</td>
</tr>
<tr>
<td>6 <strong>non-critical section</strong></td>
</tr>
</tbody>
</table>

• Where is the problem?
Problem with the first attempt

- Line 1 may not be executed atomically
- Hence two processes may get the same ticket number
- Then a deadlock can happen in line 3, as none of the processes' ticket numbers is less than the other
A suggestion for a fix

- Replace the comparison `number[i] < number[j]` by `(number[i], i) < (number[j], j)`
- The "less than" relation is defined in this case as

  \[(a, b) < (c, d) \quad \text{if} \quad (a < c) \text{ or } ((a = c) \text{ and } (b < d))\]

- **Idea:** if two ticket numbers turn out to be the same, the process with the lower identifier gets precedence
The fix doesn't work

• Unfortunately, with the fix we no longer have mutual exclusion:
  • P1 and P2 both compute the current maximum as 0
  • P2 assigns itself ticket number 1 (number[2] := 1) and proceeds into critical section
  • P1 assigns itself ticket number 1 (number[1] := 1) and proceeds into critical section, because (number[1], 1) < (number[2], 2)
Finally, we make sure that no process can read from number[j] in line 3 while process $P_j$ is still calculating its ticket number.

```
number[1] := 0; ...; number[n] := 0

$P_i$

1. choosing[i] := true
2. number[i] := 1 + max(number[1], ..., number[n])
3. choosing[i] := false
4. for all j != i do
   5.   await choosing[j] = false
   6.   await number[j] = 0 or (number[i], i) < (number[j], j)
end
7. critical section
8. number[i] := 0
9. non-critical section
```
Two lemmas

Lemma 1. If processes $P_i$ and $P_k$ are in the bakery and $P_i$ entered the bakery before $P_k$ entered the doorway, then $\text{number}[i] < \text{number}[k]$.

Lemma 2. If process $P_i$ is in its critical section and process $P_k$ is in the bakery then $(\text{number}[i], i) < (\text{number}[k], k)$. For $P_i$, choosing[$k$] = false when reading it in line 5

If we have the situation of Lemma 1, we are finished.

If $P_k$ had left the doorway before $P_i$ read $\text{number}[k]$, it was reading its current value.

Since process $P_i$ did not execute line 5 again for $j = k$, it must have found $(\text{number}[i], i) < (\text{number}[k], k)$.
Correctness of the bakery algorithm

• The Bakery algorithm satisfies mutual exclusion.  
  Proof. Follows from Lemma 2.

• The Bakery algorithm is deadlock-free.  
  Proof. Some waiting process $P_i$ has the minimum value of
  $(\text{number}[i], i)$ among all the processes in the bakery. This
  process must eventually complete the for loop and enter
  the critical section.

• The Bakery algorithm is first-come-first-served.  
  Proof. Follows from Lemmas 1 and 2.
Unbounded ticket numbers

• *Drawback of the Bakery algorithm:* values of the ticket numbers can grow unboundedly
  • Assume P1 gets ticket number 1 and proceeds to its critical section.
  • Then process P2 gets ticket number 2, lets P1 exit from its critical section and enters its own critical section.
  • As P1 tries to re-enter its critical section it draws ticket number 3.
  • In this manner two processes could alternatingly draw ticket numbers until the maximum size of an integer on the system is reached.
Space bounds for synchronization algorithms

- Size and number of shared memory locations is an important measure to compare synchronization algorithms.
- For Peterson’s algorithm, we count $2n - 1$ registers (bounded by $n$), and in the case of the Bakery algorithm $2n$ registers (unbounded in size).
- Large overhead: can we do better?
- One can prove in general a lower bound: mutual exclusion problem for $n$ processes satisfying mutual exclusion and global progress needs to use $n$ shared one-bit registers.
- The bound is tight (Lamport’s one bit algorithm).
Non-atomic memory access

- The mutual exclusion problem makes the assumption that memory accesses are executed atomically.
- This might not be a valid assumption on multiprocessor systems, leading to inconsistencies.
- The Bakery algorithm can help here as well: each memory location is only written by a single process, hence conflicting write operations cannot occur.
Other atomic primitives (1)

- Having only atomic read and write to implement locks makes efficient implementation difficult
- Where available, locks can be built from more complex atomic primitives

\[
\text{test-and-set}(x, \text{value}) \\
do \\
\quad \text{temp} := x \\
\quad x := \text{value} \\
\quad \text{result} := \text{temp} \\
\text{end}
\]
Other atomic primitives (2)

- Using more powerful primitives, concise solutions to the mutual exclusion problem can be obtained:

\[
\begin{array}{|c|}
\hline
b := false \\
\hline
P_i \\
\hline
1 & await not test-and-set(b, true) \\
2 & critical section \\
3 & b := false \\
4 & non-critical section \\
\hline
\end{array}
\]
Other atomic primitives (3)

fetch-and-add \( (x, \text{value}) \)

\begin{verbatim}
    do
        temp := x
        x := x + value
        result := temp
    end
\end{verbatim}

compare-and-swap \( (x, \text{old}, \text{new}) \)

\begin{verbatim}
    do
        if x = old then
            x := new; result := true
        else
            result := false
        end
    end
\end{verbatim}