Concepts of Concurrent Computation

Bertrand Meyer
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
Locks

- Synchronization mechanisms based on the ideas of entry- and exit-protocols are called **locks**
- They can typically be implemented as a pair of functions:

```plaintext
lock
do
  entry protocol
end

unlock
do
  exit protocol
end
```
Busy waiting

• We will use the following statement in pseudo code
  
  \text{await} \ b

which is equivalent to

  \text{while not} \ b \ \text{loop end}

• This type of waiting is called \textit{busy waiting} or "\textit{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
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.
Solution attempt I

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

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>enter1 := true</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

```
turn := 1 or turn := 2

<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>await turn = 1</td>
<td>await turn = 2</td>
</tr>
<tr>
<td>critical section</td>
<td>critical section</td>
</tr>
<tr>
<td>turn := 2</td>
<td>turn := 1</td>
</tr>
<tr>
<td>non-critical section</td>
<td>non-critical section</td>
</tr>
<tr>
<td>end</td>
<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:
\[ G \neg (P_1 \triangleright 2 \land P_2 \triangleright 2) \]
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:
\[ G ((P_1 \triangleright 1 \land P_2 \triangleright 1) \rightarrow F (P_1 \triangleright 2 \lor P_2 \triangleright 2)) \]
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 (for two processes)

- 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
enter2 := false
turn := 1 or turn := 2

P1 | P2
---|---
while true loop
  1 | enter2 := true
  2 | turn := 1
  3 | await not enter1 or turn = 2
  4 | critical section
  5 | enter2 := false
  6 | non-critical section
end
end
```
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 $\text{enter1} = \text{true}$, and P2 must thus have seen $\text{turn} = 2$ upon entering its critical section
- P2 could not have executed line 2 after P1 entered, as this sets $\text{turn} = 1$ 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 $\text{enter2} = \text{true}$ and $\text{turn} = 1$, although P2 should have seen $\text{turn} = 2$
- 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>for $j = 1$ to $n - 1$ do</td>
</tr>
<tr>
<td>2</td>
<td>enter[$i$] := $j$</td>
</tr>
<tr>
<td>3</td>
<td>turn[$j$] := $i$</td>
</tr>
<tr>
<td>4</td>
<td>await (for all $k \neq i$: enter[$k$] &lt; $j$) or turn[$j$] != $i$</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td>5</td>
<td>critical section</td>
</tr>
<tr>
<td>6</td>
<td>enter[$i$] := 0</td>
</tr>
<tr>
<td>7</td>
<td>non-critical section</td>
</tr>
</tbody>
</table>
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
- $\text{enter}[i]$: stage the process $P_i$ is currently in
- $\text{turn}[j]$: 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 $j = \Rightarrow$
  - 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 $\nRightarrow$ bounded waiting
- bounded waiting $\nRightarrow$ 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>P2</td>
<td></td>
<td>... enters + leaves critical section</td>
</tr>
<tr>
<td>P2</td>
<td>2</td>
<td>enter[2] := 1</td>
</tr>
<tr>
<td>P3</td>
<td></td>
<td>... enters + leaves critical section</td>
</tr>
</tbody>
</table>

• 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 for all $j != i$ do</td>
</tr>
<tr>
<td>3   await number[j] = 0 or number[i] &lt; number[j]</td>
</tr>
<tr>
<td>4 end</td>
</tr>
<tr>
<td>5 critical section</td>
</tr>
<tr>
<td>6 non-critical section</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

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

- **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 \((\text{number}[2] := 1)\)
    and proceeds into critical section
  - P1 assigns itself ticket number 1 \((\text{number}[1] := 1)\)
    and proceeds into critical section, because
    \((\text{number}[1], 1) < (\text{number}[2], 2)\)
The bakery algorithm

- Finally, we indicate with a flag if a process is currently calculating its ticket number

```plaintext
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 number[$k$], it was reading its current value.
Since process $P_i$ went on into the critical section, 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

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

• Note that \( x \) in this pseudo-code is treated as a reference
Other atomic primitives (2)

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

<table>
<thead>
<tr>
<th>b := false</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_i$</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Other atomic primitives (3)

fetch-and-add (x, value)
  do
    temp := *x
    *x := *x + value
    result := temp
  end

compare-and-swap (x, old, new)
  do
    if *x = old then
      *x := new; result := true
    else
      result := false
    end
  end