Critical Section Problem

Ch 3 [BenA 06]

Critical Section Problem
Solutions without HW Support
State Diagrams for Algorithms
Busy-Wait Solutions with HW Support

Mutual Exclusion

Real World Example

• How to reserve a laundry room?
  – Housing corporation with many tenants
• Reliable
  – No one else can reserve, once one reservation
    for given time slot is done
  – One can not remove other’s reservations
• Reservation method
  – One can make decision independently (without discussing with
    others) on whether laundry room is available or not
  – One can have reservation for at most one time slot at a time
  – no simultaneous resource possession
• People not needing the laundry room are not bothered
• One should not leave reservation on when moving out
• One should not lose reservation tokens/keys

3.11.2008 Copyright Teemu Kerola 2008
Concurrent indivisible operations

- Echo
  ```
  char out, in; // globals
  procedure echo {
    input (in, keyboard);
    out = in;
    output (out, display);
  }
  ```
  - What if `out` and/or `in` local variables?

- Data base update
  - Name, id, address, salary, annual salary, …

- How/when/by whom to define granularity for indivisible operations?
Critical Section (CS)

- Mutex (mutual exclusion) solved
- No deadlock: someone will succeed
- No starvation (and no unnecessary delay)
  - Everyone succeeds eventually
- Protocol does not use common variables with CS actual work
  - Can use its own local or shared variables

Algorithm 3.1: Critical section problem

<table>
<thead>
<tr>
<th>local variables</th>
<th>global variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>p</td>
</tr>
<tr>
<td>non-critical section</td>
<td>q</td>
</tr>
<tr>
<td>critical section</td>
<td></td>
</tr>
<tr>
<td>postprotocol</td>
<td></td>
</tr>
</tbody>
</table>

Critical Section Assumptions

- Preprotocol and postprotocol have no common local/global variables with critical/non-critical sections
  - They do not disturb/affect each other
- Non-critical section may stall or terminate
  - Can not assume it to complete
- Critical section will complete (will not terminate)
  - Postprotocol eventually executed once critical section is entered
- Process will not terminate in preprotocol or postprotocol (!!!)
Critical Section Solution

Algorithm 3.2: First attempt

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: await turn = 1</td>
<td>q2: await turn = 2</td>
</tr>
<tr>
<td>p3: critical section</td>
<td>q3: critical section</td>
</tr>
<tr>
<td>p4: turn ← 2</td>
<td>q4: turn ← 1</td>
</tr>
</tbody>
</table>

- How to prove correct? (or incorrect?)
  - Mutex? (functional correct)
  - No deadlock? (eventually someone from many will get in)
  - No starvation? (eventually specific one will get in)

Correctness Proofs

- Prove incorrect
  - Come up with one scenario that does not work
    - Two processes execute in sync?
    - Some other unlikely scenario?
    - Often non-trivial

- Prove correct
  - Heuristics: “I did not come up with any proofs (counterexample) for incorrectness and I am smart”
    - I can not prove incorrectness
      - “easy”, unreliable
    - It must be correct...
  - State diagrams
    - Describe algorithm with states:
      - relevant control pointer (cp) values,
      - relevant local/global variable values
    - Often non-trivial
  - Analyze state diagrams to prove correctness
State Diagram for Alg. 3.2

- State \{p_i, q_i, \text{turn}\}
  - Control pointer \(p_i\)
  - Control pointer \(q_i\)
  - Global variable \text{turn}
  - 1st four states
- Mutex ok
  - State \{p_3, q_3, \text{turn}\}
    - not accessible in state diagram?
- No deadlock?
  - When many processes try concurrently, one will succeed
- No starvation?
  - Whenever any (one) process tries, it will eventually succeed

Algorithm 3.2

- Create complete diagram with all accessible states
- No states
  - \{p_3, q_3, 1\}
  - \{p_3, p_3, 2\}
- I.e., mutex secured
- Problem:
  - Too many states?
  - Difficult to create
  - Difficult to analyze

(Fig. 3.1)
Corretness (3)

- Mutex?
  - Ok, no state \( \{p_3, q_3, \ldots\} \)
- No deadlock?
  - many try, one can always get in? (into a state with \( p_3 \) or \( q_3 \))
    \( \{p_2, q_1, 1\} \): P can get in
    \( \{p_2, q_2, 1\} \): P can get in
    \( \{p_2, q_1 \text{ tai } q_2, 2\} \):
      - Q can get in
    \( \{p_2, q_3 \text{ tai } q_4, 2\} \):
      - P can get in eventually
    - \( \{p_i, q_2, \ldots\} \) similarly
- No starvation?
  - One tries, it will eventually get in?
    - \( \{p_2, q_1, 2\} \)
      - Q dies (ok to die in \( q_1 \)), P will starve! **Not good!**
Reduced Algorithm for Easier Analysis

Algorithm 3.2: First attempt

\[
\begin{array}{ll}
\text{integer turn} & \leftarrow 1 \\
\hline
p & q \\
\text{loop forever} & \text{loop forever} \\
p_1: & \text{non-critical section} \\
p_2: & \text{await turn} = 1 \\
p_3: & \text{critical section} \\
p_4: & \text{turn} \leftarrow 2 \\
q_1: & \text{non-critical section} \\
q_2: & \text{await turn} = 2 \\
q_3: & \text{critical section} \\
q_4: & \text{turn} \leftarrow 1 \\
\end{array}
\]

- Reduce algorithm to reduce number of states of state diagrams: leave irrelevant code out
  - Nothing relevant (for mutex) left out?

Algorithm 3.5: First attempt (abbreviated)

\[
\begin{array}{ll}
\text{integer turn} & \leftarrow 1 \\
\hline
p & q \\
\text{loop forever} & \text{loop forever} \\
p_1: & \text{await turn} = 1 \\
p_2: & \text{turn} \leftarrow 2 \\
q_1: & \text{await turn} = 2 \\
q_2: & \text{turn} \leftarrow 1 \\
\end{array}
\]

State Diagram for Reduced Algorithm

- Much fewer states!
Correctness of Reduced Algorithm (2)

- Mutex?
  - No state \( \{ p_2, q_2, \text{turn} \} \)
- No deadlock: Some are trying, one may get in?
  - Top left (p & q trying): q will get in
  - Bottom left (p trying): q will eventually execute (assumption!)

- No starvation?
  - Tricky, reduced too much!
  - NCS combined with await
  - Look at original diagram
  - Problem if Q dies in NCS

Critical Section Solution #2

Algorithm 3.6: Second attempt

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: await wantq = false</td>
<td>q2: await wantp = false</td>
</tr>
<tr>
<td>p3: wantp = true</td>
<td>q3: wantq = true</td>
</tr>
<tr>
<td>p4: critical section</td>
<td>q4: critical section</td>
</tr>
<tr>
<td>p5: wantp = false</td>
<td>q5: wantq = false</td>
</tr>
</tbody>
</table>

- Each have their own global variable \( wantp \) and \( wantq \)
  - True when process is in critical section
- Process dies in NCS?
  - Starvation problem ok, because it’s \( want \)-variable is false
- Mutex? Deadlock?
Attempt #2 Reduced

Algorithm 3.7: Second attempt (abbreviated)

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: await wantq = false</td>
<td>q1: await wantp = false</td>
</tr>
<tr>
<td>p2: wantp ← true</td>
<td>q2: wantq ← true</td>
</tr>
<tr>
<td>p3: wantp ← false</td>
<td>q3: wantq ← false</td>
</tr>
</tbody>
</table>

- No mutex! \{p3, q3, ?\} reachable
  - Problem: p2 should be part of critical section (but is not!)

Critical Section Solution #3

Algorithm 3.8: Third attempt

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: wantp ← true</td>
<td>q2: wantq ← true</td>
</tr>
<tr>
<td>p3: await wantq = false</td>
<td>q3: await wantp = false</td>
</tr>
<tr>
<td>p4: critical section</td>
<td>q4: critical section</td>
</tr>
<tr>
<td>p5: wantp ← false</td>
<td>q5: wantq ← false</td>
</tr>
</tbody>
</table>

- Avoid previous problem, mutex ok
- Deadlock possible: \{p3, q3, wantp=true, wantq=true\}
- Problem: cyclic wait possible, both insist their turn next
  - No preemption
Concurrent Programming (RIO) 3.11.2008

Lecture 3: Mutual Exclusion

Algorithm 3.9: Fourth attempt

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: wantp ← true</td>
<td>q2: wantq ← true</td>
</tr>
<tr>
<td>p3: while wantq</td>
<td>q3: while wantp</td>
</tr>
<tr>
<td>p4: wantp ← false</td>
<td>q4: wantq ← false</td>
</tr>
<tr>
<td>p5: wantp ← true</td>
<td>q5: wantq ← true</td>
</tr>
<tr>
<td>p6: critical section</td>
<td>q6: critical section</td>
</tr>
<tr>
<td>p7: wantp ← false</td>
<td>q7: wantq ← false</td>
</tr>
</tbody>
</table>

- Avoid deadlock by giving away your turn if needed
- Mutex ok: P in p6 only if !wantq (Q is not in q6)
- Deadlock (livelock) possible:
  \( \{p3, q3, \ldots\} \rightarrow \{p4, q4, \ldots\} \rightarrow \{p5, q5, \ldots\} \)
  - Unlikely but possible!
  - **Livelock:** both executing all the time, not waiting suspended
  - Neither one advances

Algorithm 3.10: Dekker's algorithm

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: wantp ← true</td>
<td>q2: wantq ← true</td>
</tr>
<tr>
<td>p3: while wantq</td>
<td>q3: while wantp</td>
</tr>
<tr>
<td>p4: if turn = 2</td>
<td>q4: if turn = 1</td>
</tr>
<tr>
<td>p5: wantp ← false</td>
<td>q5: wantq ← false</td>
</tr>
<tr>
<td>p6: await turn = 1</td>
<td>q6: await turn = 2</td>
</tr>
<tr>
<td>p7: wantp ← true</td>
<td>q7: wantq ← true</td>
</tr>
<tr>
<td>p8: critical section</td>
<td>q8: critical section</td>
</tr>
<tr>
<td>p9: turn ← 2</td>
<td>q9: turn ← 1</td>
</tr>
<tr>
<td>p10: wantp ← false</td>
<td>q10: wantq ← false</td>
</tr>
</tbody>
</table>

- Combine 1st and 4th attempt
- 3 global (mutex ctr) variables: shared turn, semi-private want's
  - only one process writes to wantp or wantq (= semi-private)
- turn gives you the right to insist, i.e., priority
  - Used only when both want CS at the same time
Proof

- Mutex ok: P in p8 only if !wantq (Q can not be in q8)
- No deadlock, because P or Q can continue to CS from {p3, q3, ...}
- No starvation, because
  - If in {p6, ...}, then eventually {p6, q9, ...} and {..., q10, ...}
  - Next time {p3, ...} or {p4, ...} will lead to {p8, ...}

Algorithm 3.10: Dekker’s algorithm

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td>loop forever</td>
</tr>
<tr>
<td>p1: non-critical section</td>
<td>q1: non-critical section</td>
</tr>
<tr>
<td>p2: wantp ← true</td>
<td>q2: wantq ← true</td>
</tr>
<tr>
<td>p3: while wantq</td>
<td>q3: while wantp</td>
</tr>
<tr>
<td>p4: if turn = 2</td>
<td>q4: if turn = 1</td>
</tr>
<tr>
<td>p5: wantp ← false</td>
<td>q5: wantq ← false</td>
</tr>
<tr>
<td>p6: await turn = 1</td>
<td>q6: await turn = 2</td>
</tr>
<tr>
<td>p7: wantp ← true</td>
<td>q7: wantq ← true</td>
</tr>
<tr>
<td>p8: critical section</td>
<td>q8: critical section</td>
</tr>
<tr>
<td>p9: turn ← 2</td>
<td>q9: turn ← 1</td>
</tr>
<tr>
<td>p10: wantp ← false</td>
<td>q10: wantq ← false</td>
</tr>
</tbody>
</table>

Proof

- Mutex with **no HW-support needed, need only shared memory**
- Bad: complex, many instructions
  - Must execute each instruction at a time, in this order
    - Will not work, if compiler optimizes code too much!
  - In simple systems, can do better with HW support
    - Special machine instructions to help with this problem
Mutex with HW Support

- Specific machine instructions for this purpose
  - Suitable for many situations
  - Not suitable for all situations
- Interrupt disable/enable instructions
- Test-and-set instructions
  - Other similar instructions
- Specific memory areas
  - Reserved for concurrency control solutions
  - Lock variables (for test-and-set) in their own cache?
  - Different cache protocol for lock variables?
  - Busy-wait without memory bus use?

Disable Interrupts

- Environment
  - All (competing) processes on same processor
  - Not for multiprocessor systems
    - Disabling interrupts does it only for the processor executing that instruction
- Disable/enable interrupts
  - Prevent process switching during critical sections
    - Good for only very short time
    - Prevents also (other) operating system work while in CS
Test-and-set locking variables

- Environment
  - All processes with shared memory
  - Should have multiple processors
  - Not very good for uniprocessor systems (or synchronizing processes running on the same processor)
    - Wait (busy-wait) while holding the processor!
- Test-and-set machine instruction
  - Indivisibly read old value and write new value (complex mem-op)

```
Test-and-set (common, local)
local ← common ; read state
common ← 1 ; mark reserved
```

```
Test-and-set (shLock, locked);
while (locked)
  Test-and-set (shLock, locked);
-- CS --
shLock = 0;
```

Other Machine Instructions for Synchronization Problem Busy-Wait Solutions

- Test-and-set
  - Use all in busy-wait loops

```
Test-and-set (common, local)
local ← common ; read state
common ← 1 ; mark reserved
```

- Exchange
  - “read-modify-write” memory bus transaction (local in HW register)

```
Exchange (common, local)
local ← common ; swap values
```

- Fetch-and-add
  - “read-after-write” memory bus transaction may also be used

```
Fetch-and-add (common, local, x)
local ← common ; read state
common ← common+x ; add x
```

- Compare-and-swap

```
int Compare-and-swap (common, old, new)
return_val ← common
if (common == old)
  common ← new
```