Lecture 3: Mutual Exclusion

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
    - 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

• Reservation method

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?

  Process P1
  …
  …
  input (in,..); …

  Process P2
  …
  …
  input(in,..);

  … output (out,..);

  … output(out,..);

Critical Section (CS)

• Mutex (mutual exclusion) solved
• No deadlock: someone will succeed
  – 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>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop forever</td>
<td></td>
</tr>
<tr>
<td>non-critical section preprotocol</td>
<td></td>
</tr>
<tr>
<td>critical section postprotocol</td>
<td></td>
</tr>
</tbody>
</table>

Critical Section Assumptions

Algorithm 3.1: Critical section problem

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

– They do not disturb/buy 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>q1</td>
<td>q2</td>
</tr>
<tr>
<td>p2</td>
<td>p3</td>
</tr>
<tr>
<td>p4</td>
<td>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?

- **Prove correct**
  - Heuristics: "I did not come up with any proofs (counterexample) for incorrectness and I am smart"
  - I can not prove incorrectness
  - It must be correct...

- State diagrams

  - Describe algorithm with states:
    - relevant control pointer (cp) values,
    - relevant local/global variable values

  - Analyze state diagrams to prove correctness

**State Diagram for Alg. 3.2**

- State 
  - Control pointer p
  - Control pointer q
  - Global variable turn
  - 1st four states

- Mutex ok
  - State \{p3, q3, 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
  - \{p3, q3, 2\}
  - \{p3, p3, 2\}
  - I.e., mutex secured

- Problem:
  - Too many states?
  - Difficult to create
  - Difficult to analyze

**Corretness (3)**

- Mutex?
  - Ok, no state \{p3, q3, ??\}

- No deadlock?
  - many try, one can always get in

- No starvation?
  - One tries, it will eventually get in

- All states with p2
Reduced Algorithm for Easier Analysis

Algorithm 3.5: First attempt

\[
\begin{align*}
\text{loop forever} & \quad \text{loop forever} \\
\text{p. non-critical section} & \quad \text{q. non-critical section} \\
p. \text{turn } = 2 & \quad q. \text{turn } = 2 \\
p. \text{critical section} & \quad q. \text{critical section} \\
p. \text{turn } = 1 & \quad q. \text{turn } = 1
\end{align*}
\]

- 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)

\[
\begin{align*}
\text{loop forever} & \quad \text{loop forever} \\
p. \text{turn } = 1 & \quad q. \text{turn } = 2 \\
p. \text{turn } = 2 & \quad q. \text{turn } = 1
\end{align*}
\]

State Diagram for Reduced Algorithm

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 \((p & q \text{ trying})\): \(q\) will get in
  - Bottom left \((p \text{ trying})\): \(q\) will eventually execute (assumption!)
- Top & bottom right: mirror situation

Critical Section Solution #2

Algorithm 3.6: Second attempt

\[
\begin{align*}
\text{loop forever} & \quad \text{loop forever} \\
p. \text{non-critical section} & \quad q. \text{non-critical section} \\
p. \text{wantp} = \text{false} & \quad q. \text{wantq} = \text{false} \\
p. \text{wantp} = \text{true} & \quad q. \text{wantq} = \text{false} \\
p. \text{critical section} & \quad q. \text{critical section} \\
p. \text{wantp} = \text{false} & \quad q. \text{wantq} = \text{false}
\end{align*}
\]

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

Critical Section Solution #3

Algorithm 3.8: Third attempt

\[
\begin{align*}
\text{loop forever} & \quad \text{loop forever} \\
p. \text{non-critical section} & \quad q. \text{non-critical section} \\
p. \text{wantp} = \text{true} & \quad q. \text{wantq} = \text{false} \\
p. \text{critical section} & \quad q. \text{critical section} \\
p. \text{wantp} = \text{false} & \quad q. \text{wantq} = \text{false}
\end{align*}
\]

- Avoid previous problem, mutex ok
- Deadlock possible: \((p_3, q_3, \text{wantp=true, wantq=true})\)
- Problem: cyclic wait possible, both insist their turn next
  - No preemption
1. Avoid deadlock by giving away your turn if needed.
2. Mutex ok: P in p6 only if !wantq (Q is not in q6)
3. 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

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 \( \{p3, q3, \ldots\} \)
- No starvation, because
  - If in \( \{p6, \ldots\} \), then eventually \( \{p6, q9, \ldots\} \rightarrow \{p10, q10, \ldots\} \)
  - Next time \( \{p3, \ldots\} \rightarrow \{p8, \ldots\} \)

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
- Intercept 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)

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

Other Machine Instructions for Synchronization Problem Busy-Wait Solutions

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

- **Exchange**
  - Exchange (common, local)
  - local ← common ; swap values

- **Fetch-and-add**
  - 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

**Busy-Wait Loops**

- Use all in busy-wait loops

- "read-modify-write" memory bus transaction (local in HW register)

- "read-after-write" memory bus transaction may also be used