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

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?

Executing Many Processes Concurrently

- One CPU
  - Execute one process until
    - It requests a service that takes time to do
    - Some interrupt occurs and operating system gives execution turn to somebody else
    - E.g., time slice interrupt
  - Another process may still run concurrently in GPU or some other I/O controller
- Many CPU’s
  - Execute many processes always concurrently
  - Execution turn for one process may end any time (request service, or interrupt occurs)
Critical Section Problem

- Critical section (CS)
  - Code segment that only one process may be executing at a time
  - May also be set of code segments, and only one process may be executing at a time any code segment in that set
  - Not necessarily an atomic operation
  - Other processes may be scheduled, but they can not execute in (this) critical section

- Critical Section Problem (Mutex Problem)
  - How to guarantee that only one process at a time is executing critical section?

Critical Section (CS) Solution

- Mutex (mutually exclusive code solved)
- No deadlock: someone will succeed
- No starvation (and no unnecessary delay)
- Everyone succeeds eventually
- Protocol does not use shared variables with (CS or non-CS) actual work
  - Can use it's own local or shared variables for protocols

Critical Section Assumptions

Algorithm 3.1: Critical section problem

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

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

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? (functionally correct, one at a time in CS)
  - No deadlock? (eventually someone from many will get in)
  - No starvation? (eventually any specific one will get in)

“await condition” statement

- Pseudo language construct
  - “await until my turn”
- Implement somehow waiting until given condition becomes true
  - Use clever algorithms – wait in some loop, busy wait
  - Dekker, Peterson...
  - Use hardware (HW) help – special instructions & data?
    - Interrupts, lock variables with busy wait loops,
    - Use operating system (OS) – suspend process, busy wait
    - Semaphores, barrier operations, busy waits loops,
    - Implemented using HW (or those clever algorithms)
    - Use programming language utilities – wait hidden there
      - Semaphores, monitor condition variables, barrier operations, protected object when statements,
      - Implemented using OS
- Specifics discussed more later on

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
  - I 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
Concurrent Programming (RIO) 23.1.2012

Lecture 3: Mutual Exclusion

State Diagram for Alg. 3.2

- State \( \{p_i, q_i, \text{turn}\} \)
  - Control pointer \( p_i \)
  - Control pointer \( q_i \)
  - Global variable \( \text{turn} \)
  - \( 1^{st} \) four states

- Mutex ok
  - State \( \{p_3, q_3, \text{turn}\} \)
    - not accessible
    - How to prove it?

- No deadlock?
  - When many processes try concurrently, one will succeed

- No starvation?
  - Whenever any (one) process tries, it will eventually succeed

Algorithm 3.2

Correctness

- Mutex?
  - Ok, no state \( \{p_3, q_3, ??\} \)

- No deadlock?
  - many try, one can always get in
  - \( \{p_2, q_1, 1\} \): \( P \) can get in
  - \( \{p_2, q_2, 1\} \): \( P \) can get in
  - \( \{p_2, q_3 \text{ or } q_4, 2\} \): \( P \) can get in eventually
  - \( \{p_i, q_2, ?\} \) similarly

- No starvation?
  - One tries, it will eventually get in
  - \( \{p_2, q_2, 2\} \)
  - \( Q \) dies (ok to die in q1), \( P \) will starve!

Alternate Layout for Full State Diagram

All states with \( p_2 \)

Reduced Algorithm for Easier Analysis

Proven erroneous!

Algorithm 3.5: First attempt (abbreviated)

\[
\begin{array}{c|c}
\text{Proven} & \text{Algorithm 3.2: First attempt} \\
\hline
\text{erroneous} & \text{integer turn} = 1 \\
\hline
p & \text{loop forever} \\
q & \text{loop forever} \\
p_1: & \text{non-critical section} \\
p_2: & \text{wait} \text{turn} = 1 \\
p_3: & \text{critical section} \\
p_4: & \text{turn} = 2 \\
\hline
\end{array}
\]

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)

\[
\begin{array}{c|c}
\text{Algorithm 3.5: First attempt (abbreviated)} & \text{integer turn} = 1 \\
\hline
p & \text{loop forever} \\
q & \text{loop forever} \\
p_1: & \text{wait} \text{turn} = 1 \\
p_2: & \text{turn} = 2 \\
\hline
\end{array}
\]

Many fewer states!
Correctness of Reduced Algorithm

- 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!)
    - Top & bottom right: mirror situation

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

\[ \text{Alg. 3.5} \]

OK

Not OK

should be OK to die in non-CS, but not OK to die in protocol

Critical Section Solution #2

Algorithm 3.5: Second attempt (abbreviated)

\[ \text{loop forever} \]
\[ p: \text{wait wantp = false, wantq = false} \]
\[ q: \text{wait wantp = false, wantq = false} \]

\[ \text{OK} \]

Critical Section Solution #3

Algorithm 3.8: Third attempt

\[ \text{loop forever} \]
\[ p: \text{non-critical section wantp = true, wantq = true} \]
\[ q: \text{non-critical section wantp = false, wantq = false} \]

\[ \text{OK} \]

Critical Section Solution #4

Algorithm 3.9: Fourth attempt

\[ \text{loop forever} \]
\[ p: \text{non-critical section wantp = false, wantq = false} \]
\[ q: \text{non-critical section wantp = false, wantq = false} \]

\[ \text{OK} \]

Critical Section Solution #5

Algorithm 3.10: Dekker’s algorithm

\[ \text{loop forever} \]
\[ p: \text{non-critical section wantp = true, wantq = false intturn = } \]
\[ q: \text{non-critical section wantp = true, wantq = false intturn = } \]

\[ \text{OK} \]
Proof

- Mutex ok: P in p8 only if \( \neg \text{wantq} \) (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 \( \{p6, q9, \ldots\} \) and \( \{\ldots, q10, \ldots\} \)
  - Next time \( \{p3, \ldots\} \) or \( \{p4, \ldots\} \) will lead to \( \{p8, \ldots\} \)

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 concurrent 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 run on the same processor (core?)
  - 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 (in that processor) while in CS

Test-and-set Lock 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)

Other Machine Instructions for Synchronization Problem Busy-Wait Solutions

- Test-and-set
- Exchange
- Fetch-and-add
- Compare-and-swap
Lock variables and busy wait

- Need shared memory
- Use processor while waiting
  - Waste of a processor?
  - Not so smart with just one processor
    - Busy waits suspended when time slice ends
      (i.e., when OS time slice interrupt occurs)
  - Should wait only a very short time
    - Unless plenty of processors?
  - Real fast resume when wait ends
    - Good property in some environments

Summary

- Critical section (CS)
- Critical Section Problem
- Solutions without HW Support
- State Diagrams for Algorithms
- Busy-Wait Solutions with HW Support