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 lose reservation tokens/keys
- One should not leave reservation on when moving out

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/why whom to define granularity for indivisible operations?
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 Solution

- Mutex (mutually exclusive code 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 it's own local or shared variables

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 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

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
  - 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 \( \{p_i, q_i, \text{turn}\} \)
  - Control pointer \( p_i \)
  - Control pointer \( q_i \)
  - Global variable \( \text{turn} \)

- Mutex ok
  - State \( \{p_3, q_3, \text{turn}\} \) not accessible

- 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

Correctness (3)

- 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

- \( \{p_2, q_1 \text{ or } q_2, 2\} \):
  - Q can get in

- \( \{p_2, q_3 \text{ or } q_4, 2\} \):
  - P can get in eventually

- \( \{p_i, q_2, \text{??}\} \) similarly.

- \( \{p_3, q_2, ???\} \) similarly. \( q.e.d. \)

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

- \( \{p_2, q_3, 2\} \):
  - Q dies (ok to die in q1), P will starve! Not good!

Reduced Algorithm for Easier Analysis

Algorithm 3.2: First attempt

- Loop forever
  - integer \( \text{turn} = 1 \)

- Loop forever
  - integer \( \text{turn} = 1 \)

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

Algorithm 3.5: First attempt (altered)

- Loop forever
  - integer \( \text{turn} = 1 \)

- Loop forever
  - integer \( \text{turn} = 1 \)

Much fewer states!
Correctness of Reduced Algorithm (2)

- Mutex?
  - No state \{p_2, q_2, 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
  - Problem if Q dies in NCS

Critical Section Solution #2

\begin{align*}
\text{Algorithm 3.6: Second attempt} \\
\text{Boolean wantp, wantq} \\
\text{Loop over forever} \\
\text{P:} & \text{ non-critical section} \\
\text{Q:} & \text{ non-critical section} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{critical section:} & \text{ true} \\
\text{wantp:} & \text{ false} \\
\text{wantq:} & \text{ false} \\
\end{align*}

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

Critical Section Solution #3

\begin{align*}
\text{Algorithm 3.7: Third attempt} \\
\text{Boolean wantp, wantq} \\
\text{Loop over forever} \\
\text{P:} & \text{ non-critical section} \\
\text{Q:} & \text{ non-critical section} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{critical section:} & \text{ true} \\
\text{wantp:} & \text{ false} \\
\text{wantq:} & \text{ false} \\
\end{align*}

- Avoid previous problem, mutex ok
- Deadlock possible: \{p_3, q_3, wantp=true, wantq=true\}
- Problem: cyclic wait possible, both insist their turn next
  - No preemption

Critical Section Solution #4

\begin{align*}
\text{Algorithm 3.8: Fourth attempt} \\
\text{Boolean wantp, wantq} \\
\text{Loop over forever} \\
\text{P:} & \text{ non-critical section} \\
\text{Q:} & \text{ non-critical section} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{critical section:} & \text{ true} \\
\text{wantp:} & \text{ false} \\
\text{wantq:} & \text{ false} \\
\end{align*}

- Avoid deadlock by giving away your turn if needed
- Mutex ok: P in p6 only if wantp (Q is not in q6)
- Deadlock (livelock) possible: \{p_3, q_3, \ldots\} \Rightarrow \{p_3, q_4, \ldots\} \Rightarrow \{p_5, q_5, \ldots\}
  - Unlikekly but possible!
  - Livelock: both executing all the time, not waiting suspended
  - Neither one advances

Critical Section Solution #5

\begin{align*}
\text{Algorithm 3.9: Fifth attempt} \\
\text{Boolean wantp, wantq} \\
\text{Loop over forever} \\
\text{P:} & \text{ non-critical section} \\
\text{Q:} & \text{ non-critical section} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{critical section:} & \text{ true} \\
\text{wantp:} & \text{ false} \\
\text{wantq:} & \text{ false} \\
\end{align*}

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

Dekker’s Algorithm

\begin{align*}
\text{Algorithm 3.10: Dekker’s Algorithm} \\
\text{Boolean wantp, wantq} \\
\text{Loop over forever} \\
\text{P:} & \text{ non-critical section} \\
\text{Q:} & \text{ non-critical section} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{wantp:} & \text{ true} \\
\text{while wantq:} & \text{ true} \\
\text{critical section:} & \text{ true} \\
\text{turn:} & \text{ false} \\
\end{align*}

- Combine 1st and 4th attempt
- 3 global (mutex cts) variables: shared non., semi-private want’s
  - only one process writes to want or wantq (= semi-private)
  - turn gives you the right to finish, i.e., priority
  - Used only when both want CS at the same time
Lecture 3: Mutual Exclusion

Proof

- Mutex ok: $P$ in $p_8$ only if $\neg wantQ (Q \text{ can not be in } q_8)$
- No deadlock, because $P$ or $Q$ can continue to CS from $\{p_3, q_3, \ldots\}$
- No starvation,
  - If in $\{p_6, \ldots\}$, then eventually $\{p_6, q_9, \ldots\}$ and $\{\ldots, q_{10}, \ldots\}$
  - Next time $\{p_3, \ldots\}$ or $\{p_4, \ldots\}$ will lead to $\{p_8, \ldots\}$

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 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 run on the same processor (core?)
  - Not for multiprocessor systems
- Disable 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 \textit{(busy wait)} while holding the processor?
- Test-and-set machine instruction
  - Indisputably 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