Concurrent Systems

Exercise 04 – Deadlocks

Stefan Reif

December 19, 2016

Deadlock and Livelock

Deadlocks

- Situation where resource requests can never be fulfilled [1, 2]
- Multiple requests depend on each other
  - "depend on" $\rightarrow$ delay
  - Wanted: "worst-case blocking time" [3]

Livelock

- Threads hold processor while waiting
- Hard to detect for the OS

Strategies

- Prevention
- Detection
- Crash

Countermeasures

- Mutual exclusion
  - Write lock-free code
  - ...
- Iterative resource requests
  - Atomic multi-resource requests
  - Use only a single resource?
  - ...
- No preemption
  - Temporarily de-allocate resources (e.g. during resource request)
  - Virtualization
  - ...

Recursive Mutexes

- Re-allocation is allowed for the owner
  - Nested critical sections can be hidden
    - Function calls
    - Interfaces
    - ...
  - Recursive functions
  - Interrupt transparency
    - The critical section must tolerate interrupts
    - The interrupt handler must tolerate surrounding critical sections
    - Other solutions are often better suited

- De-allocation becomes more complex
  - Nested `leave` operations must keep the mutex
  - Top-level `leave` operation releases the mutex

Global Mutex ordering

- “lower” mutex must be acquired first
  - Requires resource ranking function
  - Problems with condition variables, `join()` function, ...
- No cyclic waiting
  - Holder of m_1 waits on $m_2 \Rightarrow rank(m_1) < rank(m_2)$
  - Waiting-for graph is directed and acyclic
- Requires thread cooperation
  - Detection of ordering violations is possible ...
  - ... but how to handle such a situation?
  - Applications can deadlock if unchecked allocations exist
- Under-approximation of allowed resource allocations
  - Applications can be deadlock-free despite ordering violations

Deadlock detection

- Deadlock $\Rightarrow$ cycle in waiting-for graph
  - Such a cycle can be detected
  - Algorithm for cycle detection in graphs?
- Explicitly create the waiting-for graph
  - Bookkeeping overhead (memory, time, energy, ...)
  - Overhead even in best-case scenario
- Occasionally search for cycles
  - Too often $\rightarrow$ unnecessary overhead
  - Not often enough $\rightarrow$ Deadlock potentially not detected

Assignment 4

- Improve your LWT library
  - This assignment focuses on mutexes
- Implement recursive mutexes
  - Use a counter for nesting depth
- Implement ordered mutexes
  - Check every mutex acquisition
  - Terminate the process in the case of an invalid request
- Implement deadlock detection
  - Check all failed allocations
  - Terminate the process in the case of a deadlock

Reference List I

[1] Coffman, E. G. ; Elphick, M. ; Shoshani, A. :
System Deadlocks.
In: *ACM Computer Survey* 3 (1971), Nr. 2, S. 67–78

[2] Holt, R. C.:
Some Deadlock Properties of Computer Systems.
In: *ACM Computer Survey* 4, Nr. 3, S. 179–196

[3] Ward, B. C. ; Anderson, J. H.:
Supporting Nested Locking in Multiprocessor Real-Time Systems.
In: *Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRS 2012)*, 2012, S. 223–232