Project

We are Project 2: Bounded-timestamp register locks, Group 1

having a consensus number of infinity means that this object can be used to give a correct implementation of any object for which you can give a sequential specification. Correctness means Linearizability here.

I think, this is what the book stated about the analogue between the Turing machine and objects with consensus number infinity.

do some experiments

do some benchmarks

do not give a detailed explanation of the algorithm. dont give a proof, why it works or why it is deadlock free or whatever.

benchmarks of the projects should be run on the amd 64 core systems.

you dont need to implement everything that is presented in the paper. filter out the main idea and implement that.

you spawn some threads

let the threads share some objects

one of the performance figures is the running time

OMP\_NUM\_TREADS can control number of threads from outside the parallel region

no Q&A

no project status representation

# General Things

Benchmarks

latency - time per operation (putting something on a queue, popping something off a stack...)

throughput - number of operations per time unit (should increase with number of threads)

fairness - use some counters: how often did a thread acquire a lock etc. Be sure to use thread local counter variables. performance counters in shared arrays will have heavy impact on the performance. ("false sharing" - when operations that are logically not connected cause cache traffic).

all as functions of the number of threads

compare to **baseline** - either another implementation of the concurrent object or a sequential implementation

Do **statistics**: repeat experiment at least 30 times and give averages with confidence intervals. should be enough times such that that the throughput and latency etc. is a stable number.

There will be cases where a certain implementation will work very well and cases, where it is very bad. Try to state these cases.

Test

Do tests, define test cases. Testing does not need to be documented in the report. just do it for yourself

state the properties of the implementation

state properties of algorithms and give worst case bounds (no proofs required). The papers will make some claims: note down these claims clearly.

Memory management

dont bother. its ok to leak

# The Handin

single zip file

[name1]\_[name2]\_amp\_project\_[p.nr.]

Report: 6-10 pages excluding plots and source code

For reproducibility: State the properties of machine, compiler and environment gcc version 4.7.2 (Debian 4.7.2-5)

compile on front end. for benchmark: submit jobs using slurm.

a report with benchmarks and plots and so on

code with makefiles (maybe run it maybe not)

Projects

P1

critical sections of different sizes

this can be a parameter in the benchmark

so do something in the cs

in real memory systems, things don't always happen in the order in which it is written in the algorithm. Maybe you have to take action. Use C++ or C memory model. or use memory fences. to make sure, things happen in the correct order.

it would be interesting to run it on different memory systems. x86 is quite well behaved, so we might not see some of the problems, that we would see on an IBM system.

# Project 2: Bounded-timestamp register locks

Implement Taubenfeld, Lamport, and two out of the other three:

* Szymanski’s solution (Boleslaw K. Szymanski: A simple solution to Lamport’s concurrent programming problem with linear wait. ICS 1988: 621-626)
* Jayanti et al.’s solution (Prasad Jayanti, King Tan, Gregory Friedland, Amir Katz: Bounding Lamport’s Bakery Algorithm. SOFSEM 2001: 261-270)
* Aravind’s solution (Alex A. Aravind: Yet Another Simple Solution for the Concurrent Programming Control Problem. IEEE Trans. Parallel Distrib. Syst. 22(6): 1056-1063, 2011)
* Black-white Bakery (Gadi Taubenfeld: The Black-White Bakery Algorithm and Related Bounded-Space, Adaptive, Local-Spinning and FIFO Algorithms. DISC 2004: 56-70)
* Lamport’s Bakery (lecture version or original)

baseline: Lamport's bakery (unbounded) and Taubenfeld

with performance counters and assertions give a plausibility check of what is claimed in the paper

compare with the following locks: pthreads, native C11, simple test-and-set lock, simple test-and-test-and-set lock,

# Papers

## Lamport 1974

He presents a solution to the mutual exclusion problem, that works if any of the computers fails at any time and has to be restarted. If they fail in the critical section a deadlock can occur. In a way it also seems to be a solution to a MRSW register problem? I did not quite understand this. Maybe the critical section is the writing to that register.

The algorithm goes like this:

**begin integer** *j;  
 L1 : choosing* [i] : = 1 ;  
 *number[i]* := 1 + *maximum (number[l],..., number[N]);  
 choosing[i]* := 0;  
 **for** j = 1 **step** l **until** N **do**  
 **begin**  
 L2: **if** *choosing[j] != 0* **then** **goto** L2;  
 L3: **if** *number[j] != 0* **and** *(number [j], j) < (number[i],* i) **then goto** L3;  
 **end**;  
 *critical section;  
 number[i]* := O;  
 *noncritical section;* **goto** L1;  
**end**

I don’t understand all of it, but it looks a lot like the bakery algorithm from the lecture. Difference maybe: he is not using a flag, that says: “I am trying to get in the CS” but his flag states, that he is drawing a number. The drawn number being nonzero serves as the flag from the lecture.

He defines the doorway section. He defines the bakery section as waiting plus critical section.

## Szymanski 1988

## Jayanti 2001

## Aravind 2011

He presents two solutions to the mutual exclusion problem.

Two fairness features are considered: FIFO and LRU (Least Recently Used)

Their algorithms are applicable for weaker memory systems, where read/write operations are not atomic. Their shared variables only need to be safe.

### System model

System model: there is a shared resource R that several processes try to access. Accessing R is done in the CS. Every process works off its code, that can be divided into a CS part and a noncritical part.

Entry Section: is what he calls the lock() function

Exit Section: is what he calls the unlock() function

Terms of Lamport for shared variables (safe, regular, atomic)

Next he writes, that no two writes on the same memory location overlap (I don’t think that safe implies that, but ok. He makes it look like it would). So we are in MRSW land here.

Memory model: Sequential consistency is assumed.

Assumptions about processes failing.

Definition of Fault Tolerance: satisfying a given property in the presence of failures.

My Remark: A failure seems to be a process breaking down somewhere followed by it resetting in its noncritical section with its lock-values reset to their default state.

### LRU Algorithm

Algorithm

Make a lock that satisfies the LRU property.

The lock has a timestamp that logs for each process, when it last acquired the lock. These timestamps are assumed to be unbounded (algorithm 1) or bounded (improved algorithm 2).

The algorithm uses three arrays:

* c[j]: Boolean – indicates whether process is trying to acquire the lock
* stage[j]: Boolean – indicates the stage that process is in (see below)
* ts[j]: integer – indicates when process has last acquired the lock

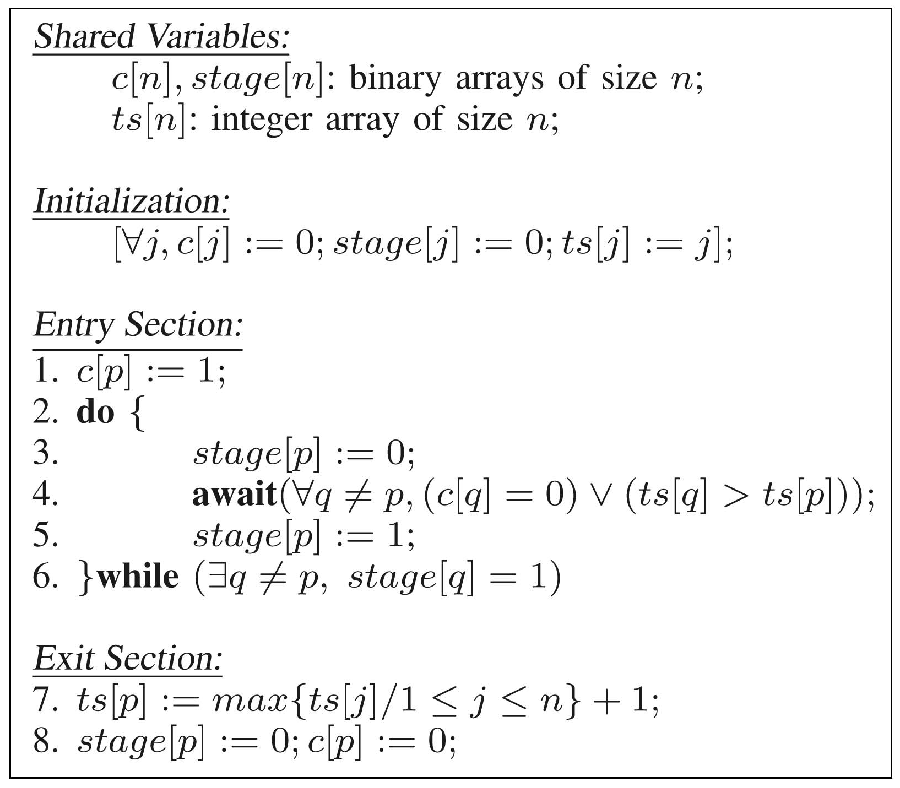
when process tries to acquire the lock it will:

* at stage 0
  + set c[j] = 1
  + check all c[k] and ts[k] to check that no one else should rather get the lock.   
    If ok, set stage[j] = 1; if not, wait and check again,
* at stage 1
  + check that no one else is at stage 1: if ok, proceed to CS, if not set stage[j] = 0 and start over

When a process unlocks, it will:

* Set ts[j] = max(ts) + 1
* Set stage[j] = 0, c[j] = 0

I am not sure, if I got the stage 1 right. The paper states, that if several processes are at stage 1, they all go back to stage 0. Then they check who of them should get to the critical section. (probably through the usual stage 0 check). This way they can be thrown back several times due to new arrivals with lesser time stamps. But looking at the pseudo-code, it seems to be like I state here.



The argument for the correctness is like this: the do/while loop will only allow at most one process to pass (correctness). The await statement makes sure only the highest priority ones will pass (fairness).

Properties

* Correctness
* freedom from deadlock
* freedom from starvation
* LRU (no process with lower priority can overtake; no process with higher priority can overtake more than once)
* Fault tolerance (My remark: seems like fault tolerance is stated for any number of faults)

Comparison to Bakery Algorithm

The tokens used for competition for the lock are calculated when performing the unlock() while in the Lamport algorithm they are calculated in the doorway section of the lock() function.

Tokens are unique while in Lamport algorithms, two processes can calculate the same token, which is considered a fairness issue by some authors.

They are both failure tolerant.

Non-atomicity. The access to the stage object is nonatomic.

There is an argument, that the LRU algorithm actually satisfies some level of FIFO (something about the global clock, that I did not understand)

Lamports algorithm requires unbound tickets to retain fairness. LRU requires unbound timestamps to retain freedom from starvation.

### BLRU Algorithm

Bounded-Timestamps-Based LRU Algorithm

Once a ts value would exceed the maximum valid value, all are reset to the initial values. This violates the LRU property. However it still holds, that no process will encounter more than one reset of its value while competing for the lock. My Remark: this implicitly seems to assume that the maximum value of is smaller than the number of processes (which is quite a safe assumption).

This implies, that no process with lower priority can overtake more than once and no process with higher priority can overtake more than twice.

* Final Corollary: a process starting to compete with priority k. The maximum number of overtakes is
* The number of overtakes possible on average is

It is stated, that while it is easily possible to reset the values of such that the LRU property remains fully intact, but it was deliberately chosen not to do so, because priority was given to the simplicity of the algorithm.

# C++ memory model

starting at https://en.cppreference.com/w/cpp/language/memory\_model

if two threads write to the same location we got a data race, unless we use

* signal handler
* std::atomic
* std::memory\_order
* std::mutex

in case of a data race, the program behavior is undefined.