Project – Papers Docu

**Contents**

[1 Lamport 1974 1](#_heading=h.gjdgxs)

[2 Szymanski 1988 3](#_heading=h.30j0zll)

[3 Jayanti 2001 3](#_heading=h.1fob9te)

[3.1 Problem description 3](#_heading=h.3znysh7)

[3.2 Comparison to Lamport’s Algorithm 3](#_heading=h.2et92p0)

[3.3 Lamport algorithm 4](#_heading=h.tyjcwt)

[3.4 Their approach 4](#_heading=h.3dy6vkm)

[3.4.1 Algorithm with unbounded and clustered tokens 4](#_heading=h.1t3h5sf)

[3.4.2 Algorithm with bounded tokens 5](#_heading=h.2s8eyo1)

[3.5 Stated properties of the algorithm 5](#_heading=h.17dp8vu)

[4 Taubenfeld 2004 6](#_heading=h.3rdcrjn)

[5 Aravind 2011 6](#_heading=h.26in1rg)

[5.1.1 System model 6](#_heading=h.lnxbz9)

[5.1.2 LRU Algorithm 6](#_heading=h.35nkun2)

[5.1.3 BLRU Algorithm 8](#_heading=h.1ksv4uv)

[6 Lock class 9](#_heading=h.44sinio)

[7 Benchmarks 9](#_heading=h.2jxsxqh)

[7.1 Throughput 10](#_heading=h.z337ya)

[7.2 Latency 10](#_heading=h.3j2qqm3)

[7.3 Fairness 10](#_heading=h.1y810tw)

[7.4 Failure tolerance 10](#_heading=h.4i7ojhp)

# C++11 memory model

[Stackoverflow discussion about C++11 memory model](https://stackoverflow.com/questions/6319146/c11-introduced-a-standardized-memory-model-what-does-it-mean-and-how-is-it-g)

# Lamport 1974

* L. Lamport, “A New Solution of Dijkstra’s Concurrent Programming Problem,” Comm. ACM, vol. 17, no. 8, pp. 453-455, 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.

So why does he need the **choosing flag**? To make sure, number[j] is not written, while a process reads it read it because it is only a safe register. This seems to be only important, if entered before .

But what if reads choosing[i] == 0, proceeds to L3 and to read number[i] but meanwhile has entered the doorway and is writing number[i]? Does not matter, because then came later and has lower priority. Therefore will either proceed to CS, which is fine, or wait some more, which is also fine.

What about the unlock() though? There writes to number[i] without raising choosing[i]. That does not matter, because if a process reads number[i] while process unlocks, will either determine, that he can now enter the CS because of the random value that was read from the safe register, which is fine, because has left the CS. Or will determine, that he cannot enter the CS and do some more waiting which is also fine.

There is another difference: this waiting section here **checks each process only once** and waits for each process, if necessary. In the lecture version it always checks all the processes and waits till no other process has a higher priority. This here determines, if I have once determined that I have a higher priority than another process, nothing can change that, until I have acquired the lock.

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

It is a bit different from the lecture version.

# Szymanski 1988

* Szymanski’s solution (Boleslaw K. Szymanski: A simple solution to Lamport’s concurrent programming problem with linear wait. ICS 1988: 621-626)

introduces a new fairness property: linear wait. No process can execute the CS twice, while another is waiting. This means a process trying to acquire the lock, cannot be overtaken twice by another process.

Robustness: again... it was brought up by Lamport. Two kinds of failures are considered:

* Processes breaking down and resetting in the noncritical section.
* Flickering bits (Szymanski seems to assume regular registers)

In general i got the feeling that this paper is not of the quality as the others. Maybe that is because it is older. But for example Szymanski does not state what kind of registers his algorithm requires. Also this algorithm does not seem to have a wait-free doorway section.

# Jayanti 2001

* Jayanti et al.’s solution (Prasad Jayanti, King Tan, Gregory Friedland, Amir Katz: Bounding Lamport’s Bakery Algorithm. SOFSEM 2001: 261-270)

## Problem description

It is assumed, that no process fails in the entry or the exit sections. This paper states this as a general assumption in the mutual exclusion problem.

Must have properties

* Mutual exclusion
* Starvation freedom: each process in the waiting section will eventually enter the CS
* Wait-free exit: each process will complete the exit section in a bounded number of steps, regardless of the speeds of other processes.

Nice to have

* Doorway FIFIO: there is a wait-free doorway. Pi completes doorway before Pj implies that Pi enters the CS before Pj

## Comparison to Lamport’s Algorithm

Both have all properties stated in the problem description

Lamport requires unbounded registers. This one only requires registers of size 1 bit or bits.

Both require MRSW registers. This one requires one additional register.

Lamport only needs safe registers. This one requires atomic registers.

## Lamport algorithm

[initialize ∀i : 1 ≤ i ≤ n : (gettoken[i] := **false**  
 token[i] := −1)]  
1 gettoken[i] := **true**  
2 S := {token[j] | 1 ≤ j ≤ n}  
3 token[i] := max(S) + 1  
4 gettoken[i] := **false**  
 **for** j ∈ {1 ・・・ n} − {i}  
5 **wait till** gettoken[j] = **false**  
6 **wait till** (token[j] = −1) ∨ ([token[i], i)] < [token[j], j])  
7 CS  
8 token[i] := −1

Then there is a nice description of the algorithm and why it works.

## Their approach

### Algorithm with unbounded and clustered tokens

They show that the range of active tokens can increase without bound for the Lamport algorithm. The example is based on the (very improbable) case that process calculates a new token in line 3, but stops before writing it to token[i]. Then other processes finish the CS and set their tokens to -1. Now one process goes through the doorway and sets its token to 0. Now process writes his high token to token[i].

They add these lines to Lamport:

[initialize ∀i : 1 ≤ i ≤ n : (gettoken[i] := **false**  
 token[i] := −1)]  
1 gettoken[i] := **true**  
2 S := {token[j] |1 ≤ j ≤ n}  
a x := X  
3 token[i] := max(S + {x}) + 1  
4 gettoken[i] := **false**  
 **for** j ∈ {1 ・・・ n} − {i}  
5 **wait till** gettoken[j] = **false**  
6 **wait till** (token[j] = −1) ∨ ([token[i], i)] < [token[j], j])  
b X := token[i]  
7 CS  
8 token[i] := −1

So now, the processes cannot reset the token number to 0. They show, that now the range of tokens is now bounded.

I don’t know what this is supposed to show us. It makes sure, that nobody ever draws a small token again. But when would that have happened anyways? Once there is no one competing for the lock and unlock() is executed, all tokens = -1. The next token drawn would be 0. Why is that bad?

### Algorithm with bounded tokens

omfg...

it takes them one whole page just to *define* a new max function and a new comparison of tokens.

But it is still better than Szymanski, so here we go:

***new max function***

let denote

let denote

let

then define

My Remark: I guess this definition works, because we always determine the maximum of a bunch of tokens plus the x value. So we always determine the max of some set.

***new comparison relation***

the symbol is used \prec

define

My Remark: seems weird, because in both terms is used.

Like this, the algorithm with bounded tokens is the same as the one with clustered tokens, only that it replaces the max function and the lesser than relation with these new definitions.

## Stated properties of the algorithm

mutual exclusion

Starvation Freedom

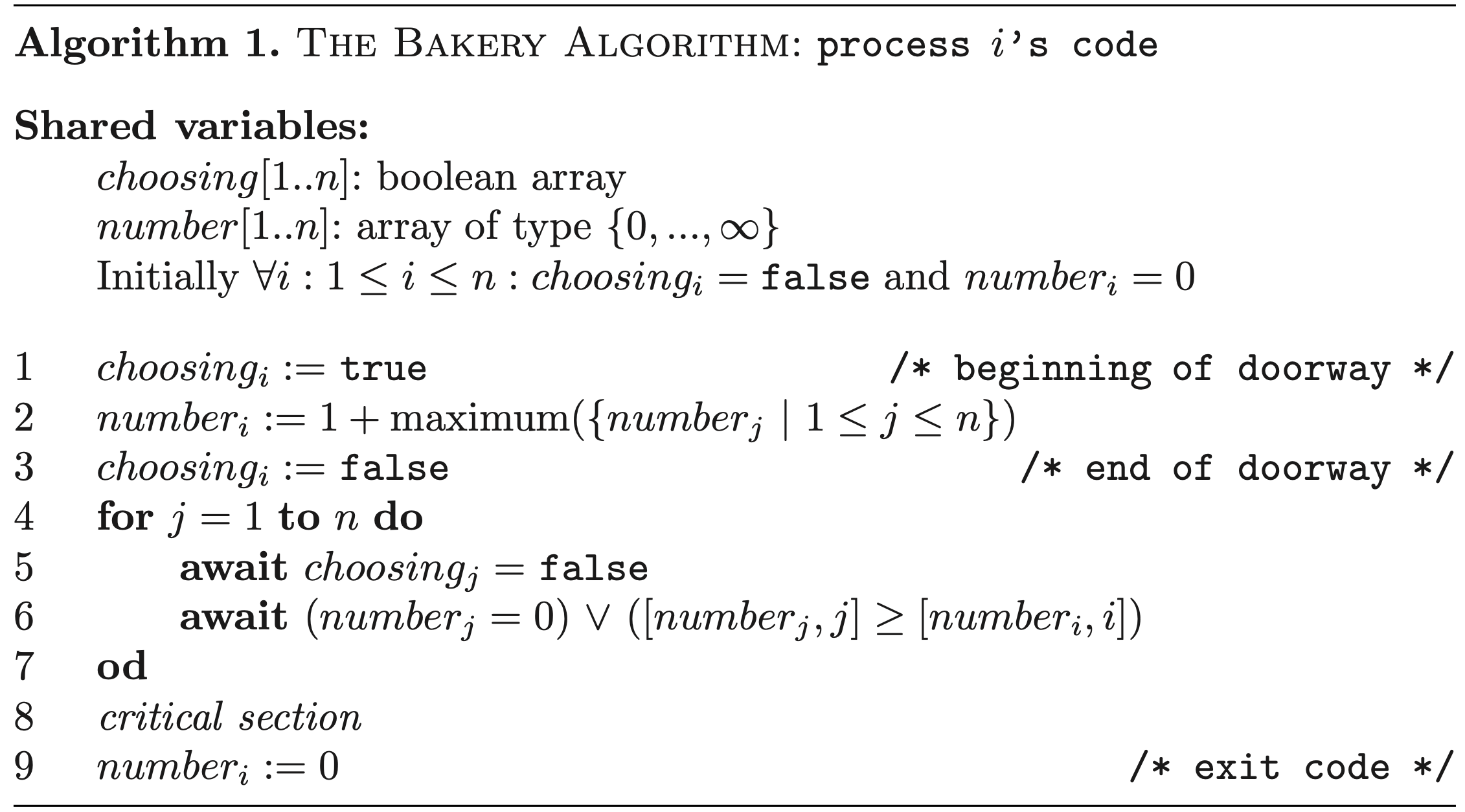
Doorway FIFO

Bounded Registers

# Taubenfeld 2004

* Black-white Bakery (Gadi Taubenfeld: The Black-White Bakery Algorithm and Related Bounded-Space, Adaptive, Local-Spinning and FIFO Algorithms. DISC 2004: 56-70)

Modified Lamports Bakery (probably original version)



## Properties

A mutual exclusion algorithm that (supposedly) has 4 properties:

1. satisfy FIFO fairness
2. satisfy local spinning
3. adaptive
4. finite number of bounded size atomic registers (in contrast to unbounded lamports)

Uses an additional "color" bit and gives each ticket a "color"

## Basic idea

Ticketing system similar to at the "Amt" in Austria:

- customer receives a ticket with a number (> number of customers waiting) AND color (black/white) upon entry.

- lowest number is served

1) FIFO is "trivial"

2) registers are bounded to size n, where n is number of processes (processes=customer in analogy)

Note: FIFO is a stronger criterium than Mutual Exclusion (ME), Deadlock Freedom(DF) and Starvation Freedom (SF) BUT does not imply SF.

## Local spinning

Spin-waiting: in wait-loop (waiting section), a flag register is "spun" (aka flipped from 0 to 1 and back?) until some \*other\* process terminates the spin. This spin flag is/was often located on some outside memory (e.g. on another node or w/e) --> possibly generates a lot of network traffic and slows other processes, as the spinning process constantly reads/writes OVER the network.

Local-spinning: a process only spins on a local register --> only 1 \*outside\* write from another process required to terminate

## Adaptive

*An algorithm is adaptive with respect to time complexity measure Phi, if its time complexity Phi is a function of actual number of contending processes. \*From Taubenfeld paper*

Later in paper:

*Time complexity: The maximum number of remote accesses which cause com- munication that a process, say p, may need to perform in its entry and exit sec- tions in order to enter and exit its critical section since p started executing the code of its entry section.*

Adaptive + local spinning = scalable algorithm

Note: Has something to do with contention of different processes and system response time...will see, I don’t quite understand this yet.

## Procedure

- Take ticket\_i = (color\_i, number\_i): read shared bit \*color\* and set color\_i=color. Set number\_i > all other tickets with same color.

- Entry to CS is given to "lowest" ticket:

1. Tickets with color\_i != \*color\* are lower than color\_i == \*color\*.

2. If 2 tickets have same color, the ticket with lower number is lower.

3. If 2 tickets are equal, the process with the lower PID is lower

- When a process i leaves the CS, it sets \*color\* != color\_i (to a different color than its own ticket color). This way, priority is given to tickets with same color as i!

Example:

\*color\* = white

- Threads 1, 2, 5 (in that order) take tickets colored white

- Thread 1 completes CS

\*color\* = black

- Threads 3, 4+1 (in that order, 4 and 1 take same number --> 1<4) take tickets colored black

- Meanwhile: 2, 5 (in that order) execute CS --> \*color\* stays black

- Thread 3 (lowest of blacks) enters CS

- Thread 2 takes ticket colored white (1 < 4 < 2)

- Thread 3 finishes CS

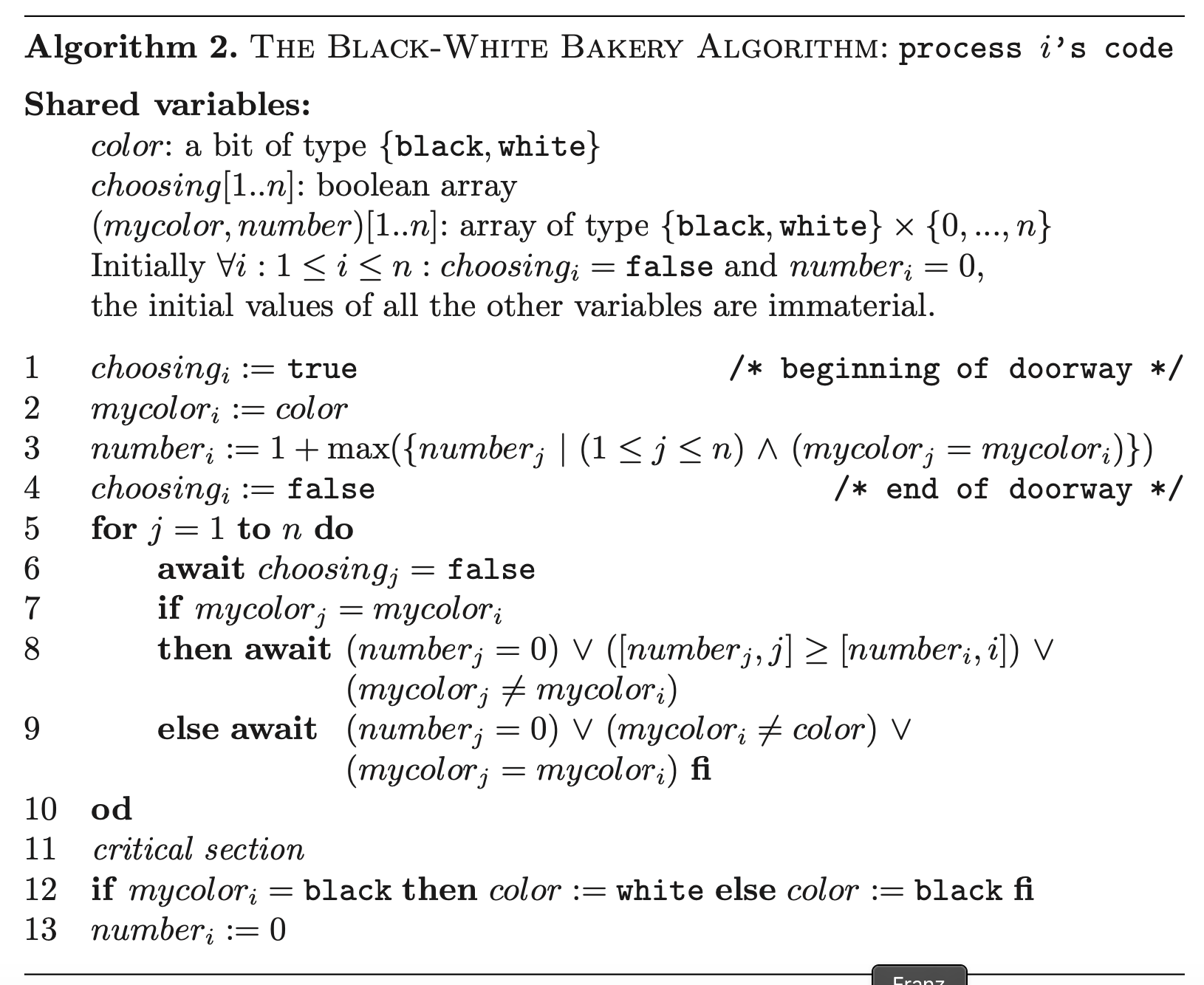
\*color\* = white

- Thread 1 enters CS

...

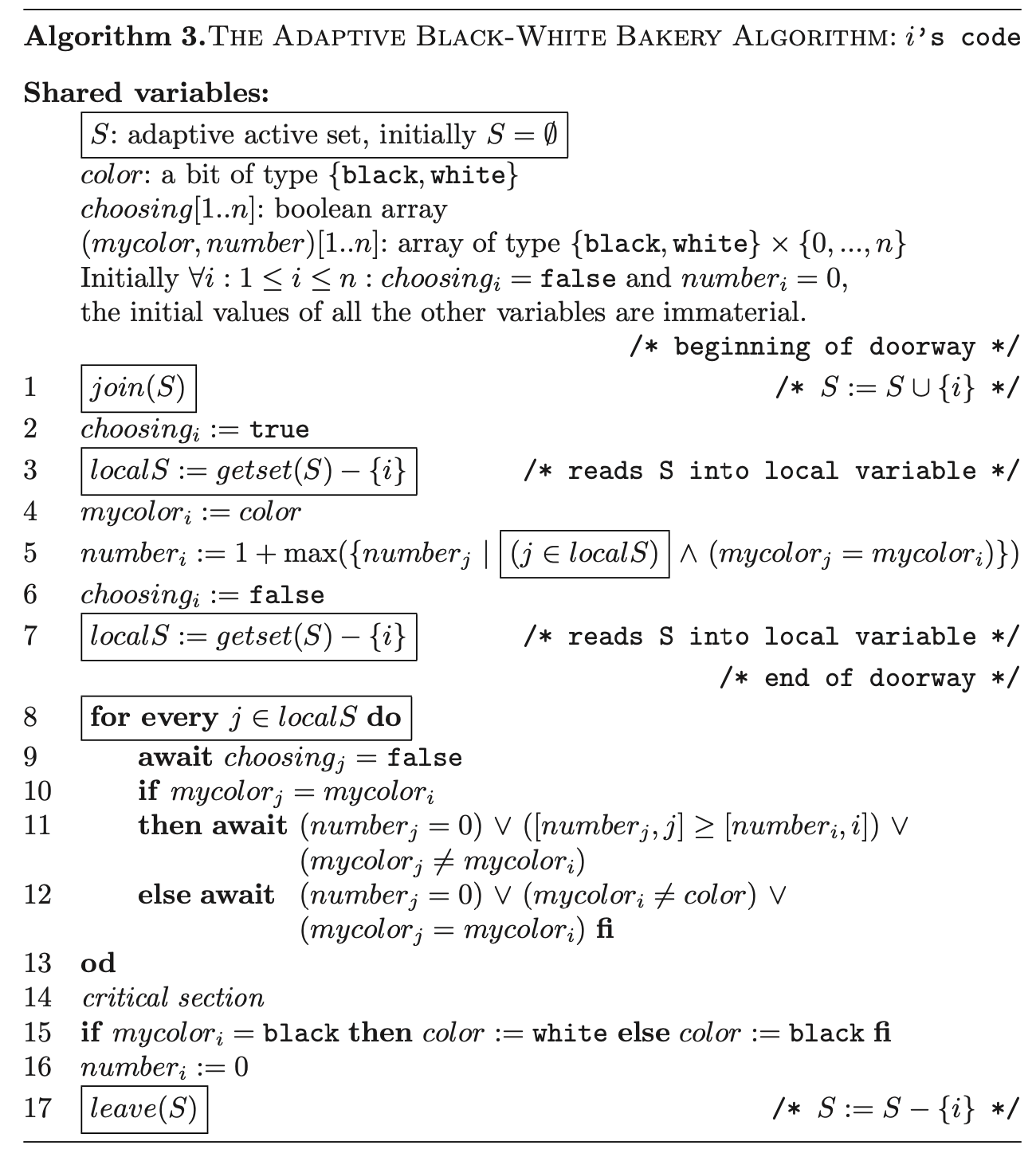
Colors are alternated in blocks, depending on when threads take their tickets.

## Pseudocode



## Transformation to Adaptive

They describe a transformation to make the algorithm adaptive (based on another paper)...but I don’t understand that yet. The transformation uses A*ctive Set* - Objects. I think, they behave like python Sets, and contain a list of currently “active“ threads (TIDs) - here this means, that they entered the doorway (they join the Set) and they leave again after executing the CS and finishing the unlock.



This one is not yet local spinning though! → There is another version in the paper, at the end.

# Aravind 2011

* 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)

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

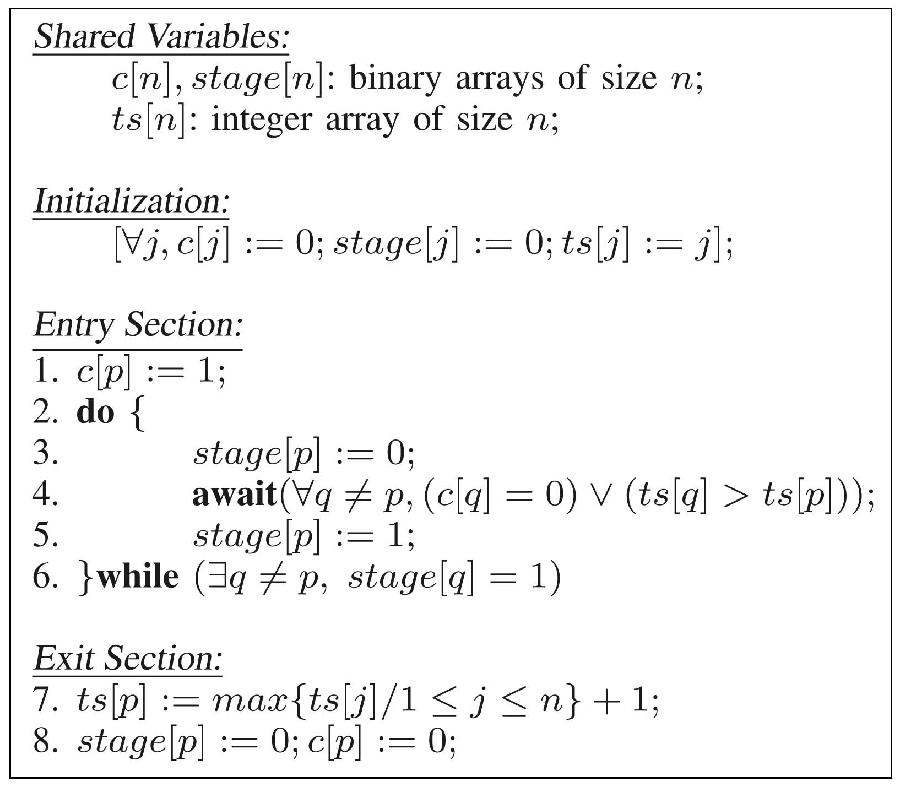
when process tries to acquire the lock it will:

* 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

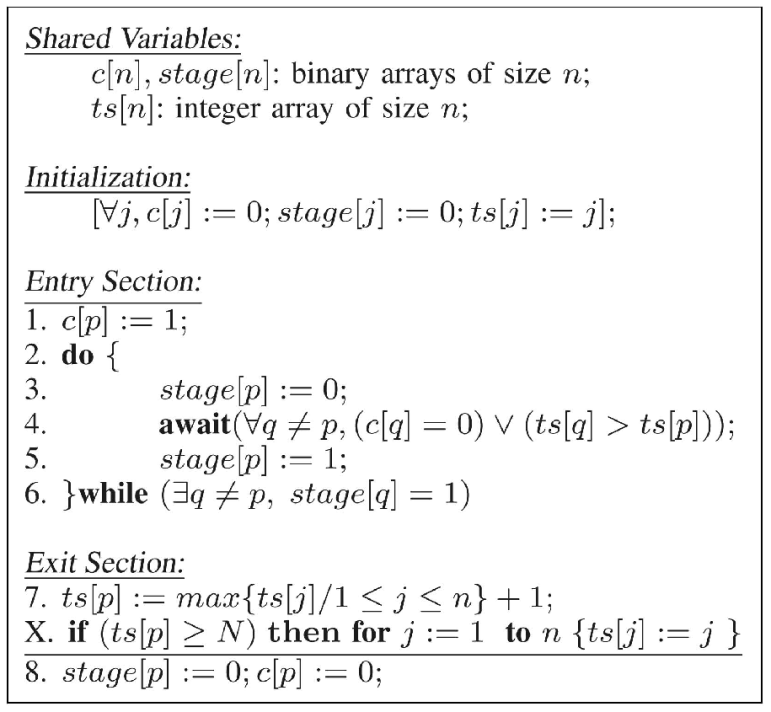
Bounded-Timestamps-Based LRU Algorithm

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.



# Lock class

size (number of threads)

lock()

unlock()

question: should we split lock() into doorway() und wait() (maybe we can test doorway FIFO)

subclasses Lamport, Taubenfeld, Jayanti, Aravind,

# Benchmarks

Parameters

num\_threads ... number of threads

cs\_workload ... how much work a thread has to do inside the critical section

ncs\_workload ... how much work a thread has to do outside the critical section

workload\_randomness ... parameter, how much randomness should be applied on the workload

This relation of these parameters defines, how much the lock is contended: the higher the values of cs\_workload and num\_threads, the more the lock will be contended. The higher the value of ncs\_workload the less the lock will be contended.

Some randomness is applied on the workload to keep the system from repeating the same sequence over and over.

## Throughput

should there be zero workload in the cs? there must be some workload in the cs - else, the lock will not be contended, but you can do little workload in the cs.  
I think, the workload can be quite arbitrary - do some math, as you implemented already - or even just sleep() for a bit.

Then how to measure? varying the number of threads and workload in and outside cs

Maybe measure contended throughput and uncontended throughput. Then latency is just the total time over number of lock acquisitions for uncontended throughput.

## Latency

time from calling lock() until it returns (when uncontended, i guess)

## Fairness

count the number of times each thread acquired the lock

cannot test doorway FIFO, because we cannot know exactly, which thread finished the doorway first.

can we test LRU fairness?

## Failure tolerance

should we make threads fail and re-initialize?