Docu - Project 2

about my templates:

use [Ctrl]+[Shift]+[1-6] to use fonts:

font 1

font 2

# font 3

## font 4

### font 5

font 6

Here the references:

* Lamport: [1]
* Taubenfeld: [2]
* Jayanti: [3]
* Aravind: [4]
* Szymanski: [5]
* Syncrobench (Gramoli): [6]

I use Mendeley as my Reference/Bibliography Manager and the Word Plugin to insert Citations. I added all the papers so you can just copy paste the citation (if you don’t use Mendeley yourself). Bibliography is at the end

# Main

do\_some\_work()

WIP

test\_mutex()

write about random workload

this will test mutual exclusion of a passed DW\_Lock object with given parameters (more detail)

The idea is to log the events of threads entering the critical section and threads leaving the CS. Mutual exclusion holds true, if every entering event is followed by a leaving event. I.e. no entering event is immediately followed by another entering event.

test\_fcfs()

write about random workload

this will test first-come-first-served of a passed DW\_Lock object with given parameters. (more detail)

At the moment this is done by making the doorway section mutually exclusive. But this will have to change.

The idea is to write an array, that keeps track of the sequence in which threads complete the doorway and another array that keeps track of threads acquiring the lock. FCFS holds true if these arrays are identical.

main()

write about initialization

An instance of one of the lock classes is created.

The lock object is then passed to one or more testing functions.

# record\_event\_log

this runs the test and records the relevant events in an array

one global atomic counter is used

Threads are sent through the lock and try to acquire it for a fixed number of times. Thread local arrays are recorded

after the test finished, threads assemble the global event log from their thread local event logs.

## Events

Four types of events are being recorded

1. being about to begin the doorway

2. having just finished the doorway

3. having just acquired the lock

4. being about to unlock

### Inaccuracies

There is a certain inaccuracy, when recording these events. In between a thread, acquiring the lock and noting that it has just acquired the lock, other threads can record events. In the this section the consequences of these inaccuracies are being discussed.

The method of this discussion:

there is a condition based on the sequence of events.

There is a noted sequence of events and a real sequence of events.

The inaccuracy causes a difference between these sequences.

Here it is discussed, whether the condition becomes stronger or weaker. If it becomes stronger this can cause false positives when checking for violations of the property. If the condition becomes weaker it is possible that violations of the property are not found.

(1) begin

A thread notes, that it is about to begin its doorway. Other threads can cause events between a thread noting, that it will now begin its doorway, and it actually beginning the doorway. So has actually happened later than what was noted.

In the real sequence can be later than in the recorded sequence. ("recorded begins appear earlier than they are").

Relevant in fcfs: condition

In the real sequence is will be more often satisfied than in the recorded sequence. Therefore the inaccuracy makes the condition weaker because is hard to satisfy if is easy to satisfy.

(2) finish

A thread notes, that has just finished its doorway. Other threads can cause events between a thread finishing its doorway and logging it. So has actually happened earlier than what was logged. ("logged finishes appear later than they are")

Relevant in fcfs: condition

Once again: In the real sequence is will be more often satisfied than in the recorded sequence. Therefore the inaccuracy makes the condition weaker.

(3) acquire

A thread notes, that has just acquired the lock. Other threads can cause events between a thread acquiring the lock and logging the acquisition. Assuming mutual exclusion for the lock, this cannot be acquisition or unlock events. So has actually happened earlier than what was logged. ("logged acquisitions appear later than they are")

* Relevant for mutual exclusion: condition

Acquisitions appearing later than they are, can cause satisfaction of the condition where it was actually violated.

* Relevant for fcfs: condition

Acquisitions appearing later than they are can cause both wrongful satisfaction of the condition as well as false positives when checking the log.

* Relevant for LRU: condition

If mutual exclusion is assumed, the order of the acquisitions cannot be changed by this inaccuracy. Therefore only is affected, which is harder to satisfy due to the inaccuracy. Therefore the LRU condition is easier to satisfy. Therefore there might be violations of the LRU property that are not registered.

(4) unlock

A thread notes, that it is about to unlock. Other threads can cause events between a thread logging the unlock and the actual unlock. Assuming mutual exclusion for the lock, this cannot be acquisition or unlock events. So has actually happened later than what was logged. ("logged unlocks appear earlier than they are")

* Relevant for mutual exclusion: condition

Unlock events appearing earlier than they are, can cause satisfaction of the condition when it was actually violated.

# Properties

## First-Come-First-Served (FCFS)

interpretation high numbers of violations

lets say we got 4 threads each acquiring the lock 3 million times

lets say the workload in the cs is so high, that on average there are 3 contenders at each acquisition

for each such acquisition there is a

* 33% chance, that 2 violations occur
* 33% chance, that 1 violation occurs
* 33% chance, that 0 violations occur

so on average 1 violation will occur per acquisition.

if we have on average 4 contenders...

* 25% chance, that 3 violations occur
* 25% chance, that 2 violation occurs
* 25% chance, that 1 violation occur
* 25% chance, that 0 violations occur

so on average: 1.5 violation. so we should see a convergence to 18 million violations if we increase the workload in the cs

## Least Recently Used (LRU)

# Locks

## Abstract lock class

We have created two abstract classes to define the common interfaces between our implementations of the various lock types.

class Lock

this pure abstract class is the Lock interface form the Herlihy book.

class DW\_Lock

this pure abstract class is an interface for "doorway locks". These are locks where the lock() function can be written like this

public: void lock(){  
 doorway();  
 wait();  
}

This is used so that FCFS can be tested from the outside (i.e. without putting logging functions into the lock class).

## Reference Lock

class Reference\_Lock

this uses these statements

* create an instance of the lock with this statement  
  std::atomic\_flag lock\_stream = ATOMIC\_FLAG\_INIT;
* execute lock() (i.e. doorway and waiting section) with this statement  
  while ( lock\_stream.test\_and\_set() ) {}
* execute unlock() with this statement  
  lock\_stream.clear();

## Lamport’s Bakery

class Lamport\_Lecture

This is the naive straight forward implementation of Lamport Bakery lock according to the lecture notes. It is very bad.

draw\_ticket

wait

class Lamport\_Lecture\_fix

This sub-class of Lamport\_Lecture uses an atomic register latest\_ticket to keep track of the ticket that was last issued. This is done by overriding the draw\_ticket() function like this:

private: virtual int draw\_ticket() override {  
 return ++latest\_ticket;  
}

Like this mutual exclusion holds. It is questionable however, if we can still speak of a wait-free doorway section now. Consider this case: Thread tries to draw a ticket, an atomic read and an atomic write of latest\_ticket are required. If latest\_ticket it is accessed by thread at the time, and thread is stalled, this could mean that there is no maximum number of required can be given in which thread completes the doorway.

class Lamport\_Original

this is the naive straight forward implementation of the Lamport Bakery algorithm from 1974. The variable names and token values are from the paper Jayati 2004.

It is very bad

implementation of wait

## Taubenfeld (Black/White-Bakery)

In 2004, Gadi Taubenfeld presented his modification[2] of Lamport’s bakery algorithm that, first and foremost, promises to bound the size of the needed registers. In total three different versions of his algorithm are presented, where the latter two build upon the first, basic version.

All three versions can be shown to satisfy *mutual exclusion*, *deadlock freedom* and *FIFO*, which also implies *starvation freedom*. The other two versions add the properties *adaptive* and *local-spinning*, although we did not implement them due to time constraints. Though we also want to note, that they are significantly more complex to achieve these added properties, whereby they lose the simple elegance of Lamport’s original algorithm.

The basic idea behind Taubenfeld’s Black/White-Bakery (sometimes also called Color-Bakery) is to add a color to the tickets, either black or white, and then working through a block of one color before considering tickets of the other color. Tickets are colored based on the value of a shared Color bit. Tickets with a color different to the shared Color bit are considered to have higher priority than those that share the shared bits color, regardless of the tickets’ numbers. In each color block, priority is given to the ticket with the lower number, and failing that, the threads’ IDs are considered. A thread that has successfully acquired the lock and executed the CS will afterwards set the shared Color bit to the color different to its ticket and then reset its ticket.

In this way the finite number of registers needed can be each bound to the sizes *log(2n +2) bit* and *1 bit* respectively. In total three registers are needed, one shared register of size 1 bit for the color

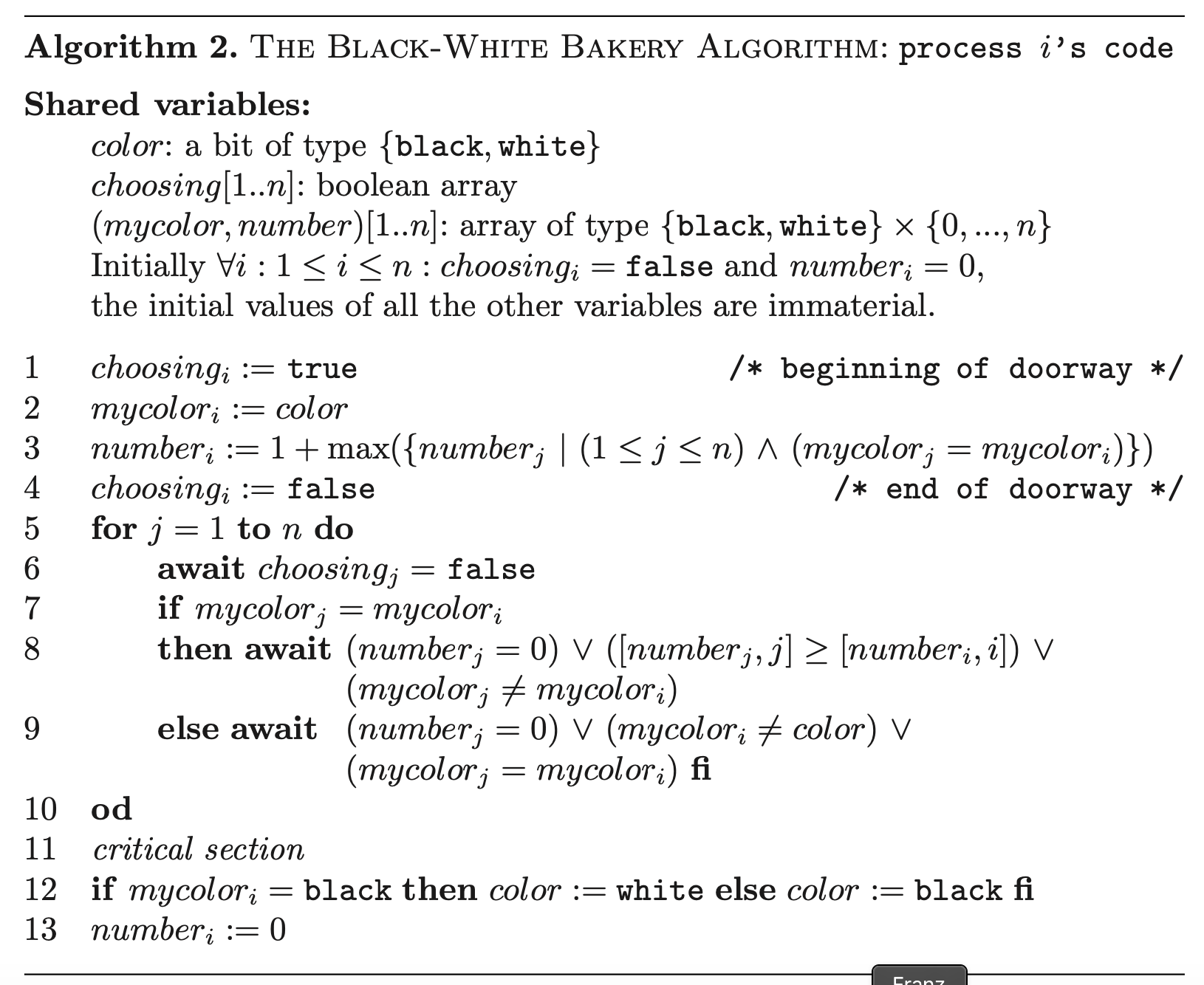


Figure 1 Black-White Bakery Algorithm[2]

class Taubenfeld

The first, basic variant presented by Taubenfeld in his paper as shown in Figure 1.

class Taubenfeld\_atomic

## Jayanti

## Aravind

# Makefile

Compiler flags

they do weird things

O1 and O3 will break mutex for Lamport\_Original

O1 and O3 will improve mutex for Lamport\_Lecture (lol)

O2 seems to do nothing (test some more?)

# References

[1] L. Lamport, “A New Solution of Dijkstra’s Concurrent Programming Problem,” *Commun. ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[2] G. Taubenfeld, “The black-white bakery algorithm and related bounded-space, adaptive, local-spinning and FIFO algorithms,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 3274, no. 4, pp. 56–70, 2004.

[3] P. Jayanti, K. Tan, G. Friedland, and A. Katz, “Bounding Lamport ’ s Bakery Algorithm,” pp. 261–270, 2001.

[4] A. A. Aravind, “Yet another simple solution for the concurrent programming control problem,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 6, pp. 1056–1063, 2011.

[5] B. K. Szymanski, “A simple solution to Lamport’s concurrent programming problem with linear wait,” *Proc. Int. Conf. Supercomput.*, vol. Part F130184, pp. 621–626, 1988.

[6] V. Gramoli, “More than you ever wanted to know about synchronization: Synchrobench, measuring the impact of the synchronization on concurrent algorithms,” *Proc. ACM SIGPLAN Symp. Princ. Pract. Parallel Program. PPOPP*, vol. 2015-January, pp. 1–10, 2015.