Distributed Algorithms
Lecture 3

Lecture by Dr. Gadi Taubenfeld
Typeset by Steven Karas

2017-04-19
Last edited 18:19:27 2017-04-19

Disclaimer These lecture notes are based on the lecture for the course Distributed Algorithms, taught by Dr. Gadi Taubenfeld at IDC Herzliyah in the spring semester of 2017. Sections may be based on the lecture slides and accompanying book written by Dr. Gadi Taubenfeld.

HW1 The first homework was published on the site, and is due on May 11th (3 weeks). It is strongly recommended to not wait to start the homework, as some of the questions require some time to think about solutions.

1 Mutual Exclusion

1.1 Review

Mutual Exclusion the property that no two processes enter the critical section.

Deadlock-Freedom the property that if n processes are blocked, at least 1 process must continue to the critical section.

Starvation-Freedom the property that if a process begins the locking code, it must eventually continue to the critical section. This implies deadlock freedom.

FIFO The property that processes execute the critical section in the order that they arrived at the locking code.

1.2 Bakery algorithm

Invented by Leslie Lamport in 1973, published in 1974. Provides mutex and FIFO. However, number[i] is unbounded. Works with safe registers, where concurrent reads may return an arbitrary value.

For processes $i = 1...n$

```
choosing[i] = true
number[i] = 1 + max(number[j]|1 ≤ j ≤ n)
choosing[i] = false
for j in 1...n:
  await choosing[j] = false
  await (number[j] = 0) or (number[j], j) ≥ (number[i], i)

CRITICAL SECTION

number[i] = 0
```

1.2.1 Finding the max

For processes $i = 1...n$

```
number[i] = 1 + max(number[j]|1 ≤ j ≤ n)
```

```
local1 = 0
for local2 in 1...n:
  local3 = number[local2]
  if local1 < local3:
    local1 = local3

number[i] = 1 + local1
```

A possible solution If we replace the $<$ with $\leq$, this may work, however in many years no one has provided a counterexample.

1.3 Black white bakery

Invented by Dr. Gadi Taubenfeld in 2004. Provides FIFO, in bounded space + one bit.

For processes $i = 1...n$

```
choosing[i] = true
mycolor[i] = color
number[i] = 1 + max{number[j]|(1 ≤ j ≤ n) ∧ (mycolor[j] = mycolor[i])}
choosing[i] = false
for j in 0...n:
  await choosing[j] = false
  if mycolor[j] = mycolor[i]:
    await (number[j] = 0) or (number[j], j) ≥ (number[i], i) or mycolor[j] ≠ mycolor[i]
  else:
    await (number[j] = 0) or mycolor[j] ≠ color or mycolor[j] = mycolor[i]

CRITICAL SECTION

if mycolor[i] = black:
  color = white
else:
  color = black
number[i] = 0
```

Single writer bits There is a question if the multi-writer bit for the color can be replaced with single-writer bits (using parity). The answer is on page 57 of the book.

1.4 Lower bound on space usage

Theorem: Any deadlock-free mutual exclusion algorithm for n processes using only SWMR registers must use at least n such registers.

Proof: Before entering its critical section, a process must write at least once; ... ■

Note that it is not possible to do better with MWMR registers.

1.5 Upper bound on space usage

Theorem: There is a deadlock-free mutual exclusion algorithm for n processes that uses n shared bits.

The One-bit algorithm Provides mutual exclusion and deadlock freedom. Does not prevent starvation. It is not fast, nor is it symmetrical. However, it is space optimal.

For processes $i = 1...n$

```
until b[i] = true:
  b[i] = true; j = 1;
  while (b[i] = true) and (j < i):
    if b[j] = true:
      b[i] = false; await b[j] = false
    j += 1

for j = i+1...n:
  await b[j] = false

CRITICAL SECTION

b[i] = false
```

1.6 Binary Semaphores

down(S) If S $>$ 0, then S = 0. Otherwise, the process is blocked until the value becomes greater than 0. Testing and decrementing are executed atomically together.

up(S) S = 1.

1.6.1 Types of semaphores

unfair - lacks starvation freedom

weak - starvation freedom

strong - FIFO

1.6.2 Deadlock-freedom with semaphores

For processes $i = 1...n$

```
down(S)
CRITICAL SECTION
up(S)
```

1.6.3 Starvation-freedom with semaphores

This is done in constant space. The basic idea is the first process to lock the semaphore redirects all future processes to the other semaphore, and waits until all the other processes waiting on its semaphore have finished their work in the critical section. [1]

For processes $i = 1...n$

```
myqueue = queue
down(S.myqueue)
if queue = myqueue:
  otherqueue = 1 - myqueue
  down(S.otherqueue)
  queue = otherqueue
  empty = false
  until empty:
    empty = true
    up(S.myqueue)
    down(S.myqueue)

  CRITICAL SECTION

  up(S.otherqueue)
else:
  empty = false
  CRITICAL SECTION

up(S.myqueue)
```

2 Exam structure

He mentioned that the exam is typically 4 questions.

3 Next Week

???

[1] Gadi mentioned that analyzing this algorithm with some minor changes is a typical HW/Exam problem.