1 Aravind (BLRU)

Aravind's algorithm is presented in [2]. It is based on Lamport's Bakery algorithm but does not require unbounded registers. Threads have three registers each: *flag* and *stage* are bits, *date* is an integer register from 0 to N. The bits are intialized to 0, the *date* of a thread with id i is initialized at i. There are n threads, with id's 0 to $n-1$. The maximum bound for the *date* registers should be set at $N \geq 2n-1$.

Algorithm 1 Aravind's BLRU algorithm

$\mathit{flag}[i] \leftarrow 1$
repeat
    $\mathit{stage}[i] \leftarrow 0$
    await $\forall_{j \neq i} : (\mathit{flag}[j] = 0 \vee \mathit{date}[i] < \mathit{date}[j])$
    $\mathit{stage}[i] \leftarrow 1$
until $\forall_{j \neq i} : \mathit{stage}[j] = 0$
critical section
$\mathit{date}[i] \leftarrow \max(\mathit{date}[0], ..., \mathit{date}[n-1]) + 1$
if $\mathit{date}[i] \geq N$ **then**
    $\forall_{j \in [0 \ldots n-1]} : \mathit{date}[j] \leftarrow j$
$\mathit{stage}[i] \leftarrow 0$
$\mathit{flag}[i] \leftarrow 0$

2 Attiya-Welch

These algorithms are presented for 2 threads. In the pseudocode, i refers to the thread's own id, j to the other thread's id. The *flag* registers are bits, the *turn* register ranges over the two thread id's i and j. All registers are initialized at 0. The original presentation from [3] is given in Algorithm 2. The variant presentation from [9] is given in Algorithm 3.

Algorithm 2 Attiya-Welch algorithm, original presentation

$\mathit{flag}[i] \leftarrow 0$
await $\mathit{flag}[j] = 0 \vee \mathit{turn} = j$
$\mathit{flag}[i] \leftarrow 1$
if $\mathit{turn} = i$ **then**
    if $\mathit{flag}[j] = 1$ **then**
        goto line 1
else
    await $\mathit{flag}[j] = 0$
critical section
$\mathit{turn} \leftarrow i$
$\mathit{flag}[i] \leftarrow 0$

Algorithm 3 Attiya-Welch algorithm, variant presentation

repeat
    $\mathit{flag}[i] \leftarrow 0$
    await $\mathit{flag}[j] = 0 \vee \mathit{turn} = j$
    $\mathit{flag}[i] \leftarrow 1$
until $\mathit{turn} = j \vee \mathit{flag}[j] = 0$
if $\mathit{turn} = j$ **then**
    await $\mathit{flag}[j] = 0$
critical section
$\mathit{turn} \leftarrow i$
$\mathit{flag}[i] \leftarrow 0$

3 Dekker

Dekker's algorithm was presented by Dijkstra in [4], we base our presentation here on [1]. The algorithm is designed for two threads, which we once again refer to as i (me) and j (other). Just like Attiya-Welch and Peterson, each thread has a bit *flag* and there is a shared bit *turn*. All registers are initialized at 0. See Algorithm 4

Algorithm 4 Dekker's algorithm

$\mathit{flag}[i] \leftarrow 1$
while $\mathit{flag}[j] = 1$ **do**
    if $\mathit{turn} \neq i$ **then**
        $\mathit{flag}[i] \leftarrow 0$
        await $\mathit{turn} = i$
        $\mathit{flag}[i] \leftarrow 1$
critical section
$\mathit{turn} \leftarrow j$
$\mathit{flag} \leftarrow 0$

4 Dijkstra

Dijkstra's algorithm is presented in [5]. Every thread has two bits: b and c. There is also the shared register k which ranges over thread id's 0 to $n-1$. While k is initialized at 0, the bits are all intialized at 1. See Algorithm 5

Algorithm 5 Dijkstra's algorithm

1: $b[i] \leftarrow 0$
2: **if** $k \neq i$ **then**
3:     $c[i] \leftarrow 1$
4:     **if** $b[k] = 1$ **then**
5:         $k \leftarrow i$
6:     **goto** line 2
7: **else**
8:     $c[i] \leftarrow 0$
9:     **for** j **from** 0 **to** $n-1$ **do**
10:         **if** $j \neq i \wedge c[i] = 0$ **then**
11:             **goto** line 2
12: **critical section**
13: $c[i] \leftarrow 1$
14: $b[i] \leftarrow 1$

5 Knuth

This algorithm was presented in [6]. Each thread has a *control* register which ranges from 0 to 2. The register k ranges over the thread id's 0 to $n-1$. All registers are initialized at 0.

Algorithm 6 Knuth's algorithm

```
1: control[i] ← 1
2: for j from k downto 0 do
3:     if j = i then
4:         goto line 12
5:     if control[j] ≠ 0 then
6:         goto line 2
7: for j from N − 1 downto 0 do
8:     if j = i then
9:         goto line 12
10:    if control[j] ≠ 0 then
11:        goto line 2
12: control[i] ← 2
13: for j from N − 1 downto 0 do
14:    if j ≠ i ∧ control[j] = 2 then
15:        goto line 1
16: k ← i
17: critical section
18: if i = 0 then
19:    k ← N − 1
20: else
21:    k ← i − 1
22: control[i] ← 0
```

6 Lamport (3 bit)

This algorithm is presented in [7]. The pseudocode is given in Algorithm 7.

This algorithm is for an arbitrary number of threads. We use id's 0 to $N-1$ when there are N threads. The j, y and f variables are private variables in the range 0 to $N-1$. The x_i, y_i and z_i registers are all Boolean variables initially set to 0.

Lamport's Three Bit Algorithm makes extensive use of cycles. A cycle, as defined in [7], is an object of the form $\langle i_0, .., i_m \rangle$ of distinct elements. Two cycles are the same if they contain the same elements in the same order except for a cyclic permutation. The first element of a cycle is its smallest element, so we take as the representative of a cycle a list where the smallest element is at index 0. An ordered cycle has all elements in order from smallest to largest, possibly only after cyclic permutation. Since our representation of a cycle is a list with the smallest element at the first position, an ordered cycle can be represented with a sorted list.

The operation ORD S takes a set S and returns the ordered cycle containing exactly the elements from S.

In the algorithm, the Boolean function $CG(v, \gamma, i_j)$ is used. Here, v is a Boolean function mapping each element in the cycle γ to either true or false, and i_j is an element from γ.

$$CG(v, \gamma, i_j) \stackrel{\text{def}}{\equiv} v(i_j) \equiv CGV(v, \gamma, i_j)$$

$$CGV(v, \gamma, i_j) \stackrel{\text{def}}{\equiv} \begin{cases} \neg v(i_{j-1}) & \text{if } j > 0 \\ v(i_m) & \text{if } j = 0 \end{cases}$$

The phrase "$i \leftarrow j$ **cyclically to** k" means that the iteration starts with $i = j$, then j gets incremented by 1, modulo the length of the cycle. This continues until $j = k$, at which point the iteration stops without executing the loop with $j = k$. $\oplus$ is used for addition modulo the length of the cycle.

Algorithm 7 Lamport's Three Bit algorithm

1: $y_i \leftarrow 1$
2: $x_i \leftarrow 1$
3: $\gamma \leftarrow \mathrm{ORD}\{i \mid y_i = 1\}$
4: $f \leftarrow \mathrm{minimum}\{j \in \gamma \mid CG(z, \gamma, j) = 1\}$
5: **for** $j \leftarrow f$ **cyclically to** i **do**
6:     **if** $y_j = 1$ **then**
7:         **if** $x_i = 1$ **then** $x_i \leftarrow 0$
8:         **goto** line 3
9: **if** $x_i = 0$ **then goto** line 2
10: **for** $j \leftarrow i \oplus 1$ **cyclically to** f **do**
11:     **if** $x_j = 1$ **then goto** line 3
12: **critical section**
13: $z_i \leftarrow 1 - z_i$
14: $x_i \leftarrow 0$
15: $y_i \leftarrow 0$

7 Peterson

This algorithm is presented for two threads in [8]. The pseudocode is given in Algorithm 8. In the pseudocode, i refers to the thread's own id, j to the other thread's id. The *flag* registers are Boolean, the *turn* register ranges over the two thread id's i and j. All registers are initialized at 0.

Algorithm 8 Peterson's algorithm

1: $\mathit{flag}[i] \leftarrow 1$
2: $\mathit{turn} \leftarrow i$
3: **await** $\mathit{flag}[j] = 0 \lor \mathit{turn} = j$
4: **critical section**
5: $\mathit{flag}[i] \leftarrow 0$

8 Szymanski

The pseudocode for the flag algorithm is shown in Algorithm 9. The flag-algorithm is presented in [10, Figure 2], but note that we have repaired an obvious typo: [10, Figure 2] erroneously has a $\wedge$ instead of a $\vee$ in line 10. All *flag* registers are initialized at 0.

Algorithm 9 Szymanski's flag algorithm

1: $\mathit{flag}[i] \leftarrow 1$
2: **await** $\forall j.\ \mathit{flag}[j] < 3$
3: $\mathit{flag}[i] \leftarrow 3$
4: **if** $\exists j.\ \mathit{flag}[j] = 1$ **then**
5:     $\mathit{flag}[i] \leftarrow 2$
6:     **await** $\exists j.\ \mathit{flag}[j] = 4$
7: $\mathit{flag}[i] \leftarrow 4$
8: **await** $\forall j < i.\ \mathit{flag}[j] < 2$
9: **critical section**
10: **await** $\forall j > i.\ \mathit{flag}[j] < 2 \vee \mathit{flag}[j] > 3$
11: $\mathit{flag}[i] \leftarrow 0$

We translate this to a 3 bit implementation using Algorithm 8.

flag	*intent*	*door_in*	*door_out*
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	1	1	1

Table 1: Translating the integer register *flag* to three Boolean registers *intent*, *door_in* and *door_out*.

Algorithm 10 Szymanski's flag algorithm implemented with bits

$intent[i] \leftarrow 1$
await $\forall j.\ intent[j] = 0 \vee door\_in[j] = 0$
$door\_in[i] \leftarrow 1$
if $\exists j.\ intent[j] = 1 \wedge door\_in[j] = 0$ **then**
    $intent[i] \leftarrow 0$
    await $\exists j.\ door\_out[j] = 1$
if $intent[i] = 0$ **then**
    $intent[i] \leftarrow 1$
$door\_out[i] \leftarrow 1$
await $\forall j < i.\ door\_in[j] = 0$
critical section
await $\forall j > i.\ door\_in[j] = 0 \vee door\_out[j] = 1$
$intent[i] \leftarrow 0$
$door\_in[i] \leftarrow 0$
$door\_out[i] \leftarrow 0$

The 3 bit linear wait algorithm is adapted from [11, Figure 1]. The pseudocode is presented in Algorithm 11. The three bits, a, w and s for each thread, are all intialized at 0.

Algorithm 11 Szymanski's 3 bit linear wait algorithm

$a_i \leftarrow 1$
for $j \leftarrow 0$ **to** $N{-}1$ **do await** $s_j = 0$
$w_i \leftarrow 1$
$a_i \leftarrow 0$
while $s_i = 0$ **do**
    $j \leftarrow 0$
    while $j < N \wedge a_j = 0$ **do** $j \leftarrow j + 1$
    if $j = N$ **then**
        $s_i \leftarrow 1$
        $j \leftarrow 0$
        while $j < N \wedge a_j = 0$ **do** $j \leftarrow j + 1$
        if $j < N$ **then** $s_i \leftarrow 0$
        else
            $w_i \leftarrow 0$
            for $j \leftarrow 0$ **to** $N - 1$ **do await** $w_j = 0$
    if $j < N$ **then**
        $j \leftarrow 0$
        while $j < N \wedge (w_j = 1 \vee s_j = 0)$ **do** $j \leftarrow j + 1$
    if $j \neq i \wedge j < N$ **then**
        $s_i \leftarrow 1$
        $w_i \leftarrow 0$
for $j \leftarrow 0$ **to** $i - 1$ **do await** $s_j = 0$
critical section
$s_i \leftarrow 0$

References

[1] K. Alagarsamy. Some myths about famous mutual exclusion algorithms. *ACM SIGACT News*, 34(3):94–103, 2003.

[2] Alex A Aravind. Yet another simple solution for the concurrent programming control problem. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):1056–1063, 2010.

[3] Hagit Attiya and Jennifer L. Welch. *Distributed computing - fundamentals, simulations, and advanced topics (2. ed.).* Wiley series on parallel and distributed computing. Wiley, 2004.

[4] Edsger W Dijkstra. Over de sequentialiteit van procesbeschrijvingen (ewd-35). ew dijkstra archive. *Center for American History, University of Texas at Austin*, 1962.

[5] Edsger W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, 1965.

[6] Donald E Knuth. Additional comments on a problem in concurrent programming control. *Communications of the ACM*, 9(5):321–322, 1966.

[7] Leslie Lamport. The mutual exclusion problem: Part II—statement and solutions. *J. ACM*, 33(2):327–348, apr 1986. `doi:10.1145/5383.5385`.

[8] Gary L. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981. `doi:10.1016/0020-0190(81)90106-X`.

[9] Cheng Shao, Jennifer L. Welch, Evelyn Pierce, and Hyunyoung Lee. Multiwriter consistency conditions for shared memory registers. *SIAM Journal on Computing*, 40(1):28–62, 2011. `arXiv:https://doi.org/10.1137/07071158X`, `doi:10.1137/07071158X`.

[10] Boleslaw K. Szymanski. A simple solution to lamport's concurrent programming problem with linear wait. In Jacques Lenfant, editor, *Proceedings of the 2nd international conference on Supercomputing, ICS 1988, Saint Malo, France, July 4-8, 1988*, pages 621–626. ACM, 1988. `doi:10.1145/55364.55425`.

[11] Boleslaw K. Szymanski. Mutual exclusion revisited. In Joshua Maor and Abraham Peled, editors, *Next Decade in Information Technology: Proceedings of the 5th Jerusalem Conference on Information Technology 1990, Jerusalem, October 22-25, 1990*, pages 110–117. IEEE Computer Society, 1990. `doi:10.1109/JCIT.1990.128275`.