5.4.1 Dekker’s Algorithm

- Software implementation of mutual exclusion
  - By Dekker, a Dutch mathematician
  - Mutual exclusion for two threads

- First version of Dekker’s algorithm
  - Succeeds in enforcing mutual exclusion
  - Uses variable to control which thread can execute

Algorithm 1

- Use a single "turn" shared variable:

```c
int turn = 1;
cobegin
  p1: while (1) {
    while (turn=2); /*wait*/
    CS1; turn = 2; program1;
  }
  //
  p2: while (1) {
    while (turn=1); /*wait*/
    CS1; turn = 1; program2;
  }
coend
```
5.4.1 Dekker's Algorithm

- Problems with the first version of Dekker's algorithm
  - While satisfying mutual exclusion requirements,
  - Busy waiting
    - Constantly tests whether critical section is available
    - Wastes significant processor time
  - Lockstep synchronization
    - Each thread can execute only in strict alternation

---

Figure 5.6 Mutual exclusion implementation – version 1 (1 of 2).

```c
1 #define threadNumber 1
3 void thread1() { 
5 startThread(); // initialize and launch both threads
7 lockSection();
9 waitAnnounce();
11 while (true) {
13 while (threadNumber == 1); // enterMutexExclusion
16 criticalSection();
18 threadNumber = 2; // exitMutexExclusion
20 // code outside critical section
21 } // end outer while
23 }
```

---

Figure 5.6 Mutual exclusion implementation – version 1 (2 of 2).

```c
22 } // end Thread 1
24 } // end Thread 2
26 void thread2() {
28 while (true) {
30 while (threadNumber == 1); // enterMutexExclusion
33 criticalSection();
36 threadNumber = 1; // exitMutexExclusion
38 // code outside critical section
40 } // end outer while
42 } // end Thread 2.
```
5.4.1 Dekker's Algorithm

• Second version
  – Removes lockstep synchronization (strict alternation)

Algorithm 2

• Use two variables to indicate intent to use the CS

```c
int c1 = 0, c2 = 0;
cobegin
  p1: while (1) {
      while (c2); /*wait*/
      c1 = 1;
      CS1; c1 = 0; program1;
  }
  p2: while (1) {
      while (c1);
      c2 = 1;
      CS2; c2 = 0; program2;
  }
cobend
```

5.4.1 Dekker's Algorithm

• Problems with the second version
  – Violates mutual exclusion
    • Thread could be preemted while updating flag variable
    • After the while tests, both threads may enter the critical section concurrently
  – Not an appropriate solution
5.4.1 Dekker's Algorithm

Figure 5.7 Mutual exclusion implementation – version 2 (1 of 3).

```c
1. #define
2. 
3. boolean t1Inside = false;
4. boolean t2Inside = false;
5. 
6. startThread1(); // initialize and launch both threads
7. 
8. ThreadT1;
9. 
10. void main() {
11. 
12. while (!done) {
13. 
14. while (t1Inside); // enterMutualExclusion
15. 
16. t1Inside = true; // enterMutualExclusion
17. 
18. // critical section code
19. 
20. t1Inside = false; // exitMutualExclusion

21. }
22. } // end outer while
23. 
24. // Thread T1
25. 
26. ThreadT2;
27. 
28. void main() {
29. 
30. while (!done) {
31. 
32. while (t2Inside); // enterMutualExclusion
33. 
34. t2Inside = true; // enterMutualExclusion
35. 
36. // critical section code
37. 
38. t2Inside = false; // exitMutualExclusion
39. 
40. // code outside critical section
41. 
42. // end outer while
43. 
44. } // end Thread T2
```

Figure 5.7 Mutual exclusion implementation – version 2 (2 of 3).

Figure 5.7 Mutual exclusion implementation – version 2 (3 of 3).
5.4.1 Dekker's Algorithm

• Third version
  – Set critical section flag before entering critical section test
  • Once again guarantees mutual exclusion

Algorithm 3

• Use two variables to indicate intent to use the CS

```c
int c1 = 0, c2 = 0;
cobegin
  p1: while (1) {
      c1 = 1;
      while (c2); /*wait*/
      CS1; c1 = 0; program1;
  }
  p2: while (1) {
      c2 = 1;
      while (c1);
      CS2; c2 = 0; program2;
  }
coend
```

5.4.1 Dekker's Algorithm

• Problems with the third version
  – Introduces possibility of deadlock
    • Both threads could set flag simultaneously
    • Both c1 and c2 may be set to 1 at the same time
  • Neither would ever be able to break out of loop
  • Not a solution to the mutual exclusion problem
5.4.1 Dekker's Algorithm

Figure 5.8 Mutual exclusion implementation – version 3 (1 of 2).

```c
1: // begin algorithm
2: boolean shareThread = false;
3: boolean enrichThread = true;
4: initialize shareThread(); // initialize and launch both threads
5: shareThread(); // initialize and launch both threads
6: while (true) {
7: 1. while (shareThread) // enter mutual exclusion
8: 2. shareThreadEnter = true; // enter mutual exclusion
9: 3. while (shareThread) // enter mutual exclusion
10: 4. critical section code
11: 5. shareThreadEnter = false; // exit mutual exclusion
12: 6. // code outside critical section
13: 7. shareThread(); // end thread 1
14: }
```

5.4.1 Dekker's Algorithm

Figure 5.8 Mutual exclusion implementation – version 3 (2 of 2).

```c
1: // end thread 1
2: return 0;
3: while (true) {
4: 1. while (shareThread) // enter mutual exclusion
5: 2. shareThreadEnter = true; // enter mutual exclusion
6: 3. while (shareThread) // enter mutual exclusion
7: 4. critical section code
8: 5. shareThreadEnter = false; // exit mutual exclusion
9: 6. // code outside critical section
10: 7. shareThread(); // end thread 1
11: }
```

5.4.1 Dekker's Algorithm

- Fourth version
  - Sets flag to false for small periods of time to yield control
Algorithm 4

```c
int c1 = 0, c2 = 0;
cobegin
p1: while (1) {
    c1 = 1;
    while (c2){
        c1 = 0; random_wait; c1=1;
    } /*wait*/
    CS1; c1 = 0; program1;
}
p2: while (1) {
    c2 = 1;
    while (c1){
        c2 = 0; random_wait; c2=1;
    } /*wait*/
    CS2; c2 = 0; program2;
} coend
```

5.4.1 Dekker's Algorithm

- Problems with the fourth version
  - Solves previous problems, introduces indefinite postponement
    - Both threads could set flags to same values at same time repeatedly
      - both repeat the while loop
      - quite low probability
    - Would require both threads to execute in tandem (unlikely but possible)
  - Unacceptable in mission- or business-critical systems
5.4.1 Dekker's Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (2 of 4).

```
while ( iWantsToEnter ) // enter Mutual Exclusion

iWantsToEnter = false; // enter Mutual Exclusion

// wait for small, random amount of time

iWantsToEnter = true;

} // end while

// critical section code

iWantsToEnter = false; // exit Mutual Exclusion

// code outside critical section

} // end outer while
```

5.4.1 Dekker's Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (3 of 4).

```
} // end Thread T2

ThreadTp:

void main()
{

while ( iDone )
{

iWantsToEnter = true; // enter Mutual Exclusion

while ( iWantsToEnter ) // enter Mutual Exclusion

iWantsToEnter = false; // enter Mutual Exclusion

// wait for small, random amount of time

iWantsToEnter = true;

} // end while
```

5.4.1 Dekker's Algorithm

Figure 5.9 Mutual exclusion implementation – version 4 (4 of 4).

```
// critical section code

iWantsToEnter = false; // exit Mutual Exclusion

// code outside critical section

} // end outer while

} // end Thread T2
```
5.4.1 Dekker's Algorithm

- Dekker's Algorithm
  - Proper solution for two-threaded mutual exclusion solution implemented purely in software with no special-purpose hardware instructions
  - Uses notion of favored threads to determine entry into critical sections
    - Resolves conflict over which thread should execute first
    - Each thread temporarily unsets critical section request flag
    - Favored status alternates between threads
  - Guarantees mutual exclusion
  - Avoids previous problems of deadlock, indefinite postponement

Algorithm 5: Dekker's Solution

```c
int favor = p1, c1 = 0, c2 = 0;
cobegin
p1: while (1) {
    c1 = 1;
    while (c2){
        if (favor == p2){
            c1 = 0;
            while (favor == p2);
            c1 = 1;
        }
    } /*wait*/
    CS1; favor = p2; c1 = 0; program1;
    p2: while (1)
    c2 = 1;
    while (c1);
    if (favor == p1){
        c2 = 0;
        while (favor == p1);
        c2 = 1;
    }
    CS2; favor = p1; c2 = 0; program2;
} coend
```

Figure 5.10 Dekker's Algorithm for mutual exclusion. (1 of 4)
5.4.1 Dekker's Algorithm

Figure 5.10 Dekker's Algorithm for mutual exclusion. (2 of 4)

```c
while (tWantsToEnter)
{
    if (favoredThread == 2)
    { / *favoredThread = false;*/
        while (favoredThread == 2) // busy wait!
            tWantsToEnter = true;
    }
    else
    { / *end while*/
        // critical section code
        favoredThread = 2;
        tWantsToEnter = false;
    }
    // code outside critical section
    } // end while
} // end thread T1
```

5.4.1 Dekker's Algorithm

Figure 5.10 Dekker's Algorithm for mutual exclusion. (3 of 4)

```c
Thread T2:

void main()
{
while (true)
{
    tWantsToEnter = true;
    while (tWantsToEnter)
    { / * favoredThread = 1 */
        if (favoredThread == 1)
        { / * favoredThread = false;*/
            while (favoredThread == 1) // busy wait!
                tWantsToEnter = true;
        }
    } / *end while*/
    // critical section code
    favoredThread = 1;
    tWantsToEnter = false;
} // end outside
```

5.4.1 Dekker's Algorithm

Figure 5.10 Dekker's Algorithm for mutual exclusion. (4 of 4)

```c
// critical section code
} // end outer while
```

5.4.1 Dekker's Algorithm
5.4.2 Peterson's Algorithm (1981)

- Previous algorithms illustrate how complicated process synchronization can be
  - Simpler algorithm for two process mutual exclusion with busy waiting
- Less complicated than Dekker’s Algorithm
  - Still uses busy waiting, favored threads
  - Requires fewer steps to perform mutual exclusion primitives
  - Easier to demonstrate its correctness
  - Does not exhibit indefinite postponement or deadlock

Peterson Solution (1981)

- Like #2 but use a “WillWait” variable to break a tie:

```c
int c1 = 0, c2 = 0, WillWait;
cobegin
p1: while (1) {
    c1 = 1;
    WillWait = 1;
    while (c2 && (WillWait==1)); /*wait*/
    CS1; c1 = 0; program1;
}
p2:while (1) {
    c2 = 1;
    WillWait = 2;
    while (c1 && (Will_wait==2));
    CS2; c2 = 0; program2;
} 
coend
```

Why Peterson’s Solution Works

- Avoiding Mutual Blocking
  - Assume p1 circle through its while loop → blocked
  - p2 may be doing one of the following things
    - (1) not trying to enter CS
      - p1 detects C2 is 0 and enters its CS
    - (2) waiting in its own while loop
      - Impossible: will_wait is either 1 or 2 (it cannot remain in 1 or 2 forever)
    - (3) repeatedly executing its own complete loop
      - Impossible: while p1 is waiting, p2 could enter the CS only if will_wait is set to 1
      - After exiting its CS, p2 will set will_wait to 2, which will not pass the while condition until p1 executes its CS
Why Peterson's Solution Works

- Mutual Exclusion
  - Assume p1 has just passed its test and is about to enter its CS
  - c1 is now 1
  - Can p2 also enter its CS at the same time?
    - (1) Case 1: p1 passed its test because c2 is 0
      - p2 is in non-CS section. If p2 tries to enter its CS, it has to set c2 to 1
      - c1 is already 1, p2 has to wait
    - (2) Case 2: p1 passed its test because will_wait is 2
      - p2 finds that c1 is already 1 and will_wait is 2, which prevents p2 from passing the test

5.4.2 Peterson's Algorithm

Figure 5.11 Peterson's Algorithm for mutual exclusion. (1 of 3)

```c
System:
3: int favoredThread = 3;
4: boolean t1WantsToEnter = False;
5: boolean t2WantsToEnter = False;
6: startThreads(); // Initialize and launch both threads
```

Figure 5.11 Peterson's Algorithm for mutual exclusion. (2 of 3)

```c
Thread 1:
19: until main();
20: { while ( true )
21: { t1WantsToEnter = true;
22: favoredThread = 1;
23: while ( t2WantsToEnter & favoredThread = 2 );
24: // critical section code
25: t1WantsToEnter = false;
26: // code outside critical section
27: // end while
28: } // end while
```

Figure 5.11 Peterson's Algorithm for mutual exclusion. (3 of 3)
5.4.2 Peterson's Algorithm

Figure 5.11 Peterson’s Algorithm for mutual exclusion. (3 of 3)

```c
30 void wlock()
31 {
32     while (tdone)
33         { 
34             tWantsToEnter = true;
35             favoredThread = -1;
36             while (tWantsToEnter && favoredThread == 1);
37             // critical section code
38             favoredThread = false;
39             // code outside critical section
40         } // end while
41     } // end while
```

5.4.3 N-Thread Mutual Exclusion: Lamport's Bakery Algorithm

- Applicable to any number of threads
  - Creates a queue of waiting threads by distributing numbered “tickets”
  - Each thread executes when its ticket’s number is the lowest of all threads
  - Unlike Dekker's and Peterson's Algorithms, the Bakery Algorithm works in multiprocessor systems and for n threads
  - Relatively simple to understand due to its real-world analog
5.4.3 N-Thread Mutual Exclusion: Lamport's Bakery Algorithm

Figure 5.12 Lamport's Bakery Algorithm. (2 of 3)

```c
73 $T$hread $R$
74 void make()
75 {
76     $t$ = threadNumber(); // store current thread number
77     while ( idone )
78     {
79          // take a ticket:
80          choosin$R$(s) = true; // begin ticket selection process
81          ticket(s) = workstation ticket() + 1;
82          choosin$R$(s) = false; // end ticket selection process
83          // wait for number to be called by comparing current
84          // ticket value to other thread's ticket value
85          for ( int i = 0; i < $n$; i++)
86          {
87              if ( $t$ == i)
88                  continue; // no need to check own ticket
89          } // end if
90     } // end while
91 }
```

5.4.3 N-Thread Mutual Exclusion: Lamport's Bakery Algorithm

Figure 5.12 Lamport's Bakery Algorithm. (3 of 3)

```c
94     // hang wait until current ticket value is highest
95     while ( choosin$R$(s) false );
96     // hang wait untill current ticket value is lowest
97     while ( ticket(s) < $\min$ ticket(s) + ticket(s) );
98     // time-order code favors smaller thread number
99     if ( ticket[1] == ticket[s] & $t$ == i )
100     // sleep until thread[s] leaves its critical section
101     while ($\min$ ticket[s] == 0); // sleep wait
102     } // end for
103 47     // critical section code
104 48     ticket[s] = 0; // exitPMuxExclusion
105 49     // code outside critical section
106 50     $\min$ ticket[s] = $\min$ ticket[s] + 1;
107     } // end while
108     } // end thread $T$