COP 4600 (Sec 1): Operating Systems

Lecture 9

Instructor : Joohan Lee
jlee@cs.ucf.edu

http://www.cs.ucf.edu/~jlee/cop4600

School of Computer Science
University of Central Florida

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:

```
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

5.4.1 Dekker's Algorithm

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

```
1  System:
2
3  int threadNumber = 1;
4
5  startThreads(); // initialize and launch both threads
6
7  Thread T1:
8
9  void main() {
10
11    while ( !done )
12    {
13       while ( threadNumber == 2 ); // enterMutualExclusion
14
15       // critical section code
16
17       threadNumber = 2; // exitMutualExclusion
18
19       // code outside critical section
20
21    } // end outer while
```


5.4.1 Dekker's Algorithm

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

```
22
23 } // end Thread T1
24
25 Thread T2:
26
27 void main() {
28
29    while ( !done )
30    {
31       while ( threadNumber == 1 );  // enterMutualExclusion
32
33       // critical section code
34
35       threadNumber = 1; // exitMutualExclusion
36
37       // code outside critical section
38
39    } // end outer while
40
41 } // end Thread T2
```


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

```
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;
    }
coend
```


5.4.1 Dekker's Algorithm

- Problems with the second version
  - Violates mutual exclusion
    - Thread could be preempted 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).

```
1  System:
2
3  boolean t1Inside = false;
4  boolean t2Inside = false;
5
6  startThreads(); // initialize and launch both threads
7
8  Thread T1:
9
10 void main() {
11
12    while ( !done )
13    {
14       while ( t2Inside ); // enterMutualExclusion
15
16       t1Inside = true; // enterMutualExclusion
17
18       // critical section code
19
20       t1Inside = false; // exitMutualExclusion
21
```


5.4.1 Dekker's Algorithm

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

```
22       // code outside critical section
23
24    } // end outer while
25
26 } // end Thread T1
27
28 Thread T2:
29
30 void main() {
31
32    while ( !done )
33    {
34       while ( t1Inside ); // enterMutualExclusion
35
36       t2Inside = true; // enterMutualExclusion
37
38       // critical section code
```


5.4.1 Dekker's Algorithm

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

```
39
40       t2Inside = false; // exitMutualExclusion
41
42       // code outside critical section
43
44    } // end outer while
45
46 } // end Thread T2
```


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

```
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).

```
1  System:
2
3  boolean t1WantsToEnter = false;
4  boolean t2WantsToEnter = false;
5
6  startThreads(); // initialize and launch both threads
7
8  Thread T1:
9
10 void main()
11 {
12    while ( !done )
13    {
14       t1WantsToEnter = true; // enterMutualExclusion
15
16       while ( t2WantsToEnter ); // enterMutualExclusion
17
18       // critical section code
19
20       t1WantsToEnter = false; // exitMutualExclusion
21
22       // code outside critical section
23
```


5.4.1 Dekker's Algorithm

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

```
24    } // end outer while
25
26 } // end Thread T1
27
28 Thread T2:
29
30 void main()
31 {
32    while ( !done )
33    {
34       t2WantsToEnter = true; // enterMutualExclusion
35
36       while ( t1WantsToEnter ); // enterMutualExclusion
37
38       // critical section code
39
40       t2WantsToEnter = false; // exitMutualExclusion
41
42       // code outside critical section
43
44    } // end outer while
45
46 } // end Thread T2
```


5.4.1 Dekker's Algorithm

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

Algorithm 4

```
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 (1 of 4).

```
1  System:
2
3  boolean t1WantsToEnter = false;
4  boolean t2WantsToEnter = false;
5
6  startThreads(); // initialize and launch both threads
7
8  Thread Tj:
9
10 void main()
11 {
12    while ( !done )
13    {
14       t1WantsToEnter = true;  // enterMutualExclusion
15
```

5.4.1 Dekker's Algorithm

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

```
16       while ( t2WantsToEnter )  // enterMutualExclusion
17       {
18          t1WantsToEnter = false;  // enterMutualExclusion
19
20          // wait for small, random amount of time
21
22          t1WantsToEnter = true;
23       } // end while
24
25       // critical section code
26
27       t1WantsToEnter = false; // exitMutualExclusion
28
29       // code outside critical section
30
31    } // end outer while
32
```


5.4.1 Dekker's Algorithm

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

```
33 } // end Thread T1
34
35 Thread T2:
36
37 void main()
38 {
39    while ( !done )
40    {
41       t2WantsToEnter = true;  // enterMutualExclusion
42
43       while ( t1WantsToEnter )  // enterMutualExclusion
44       {
45          t2WantsToEnter = false;  // enterMutualExclusion
46
47          // wait for small, random amount of time
48
49          t2WantsToEnter = true;
50       } // end while
51
```


5.4.1 Dekker's Algorithm

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

```
52       // critical section code
53
54       t2WantsToEnter = false; // exitMutualExclusion
55
56       // code outside critical section
57
58    } // end outer while
59
60 } // 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

```
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
```


5.4.1 Dekker's Algorithm

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

```
1  System:
2
3  int favoredThread = 1;
4  boolean t1WantsToEnter = false;
5  boolean t2WantsToEnter = false;
6
7  startThreads(); // initialize and launch both threads
8
9  Thread T1:
10
11 void main()
12 {
13    while ( !done )
14    {
15       t1WantsToEnter = true;
16
```

5.4.1 Dekker's Algorithm

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

```
17       while ( t2WantsToEnter )
18       {
19          if ( favoredThread == 2 )
20          {
21             t1WantsToEnter = false;
22             while ( favoredThread == 2 ); // busy wait
23             t1WantsToEnter = true;
24          } // end if
25
26       } // end while
27
28       // critical section code
29
30       favoredThread = 2;
31       t1WantsToEnter = false;
32
33       // code outside critical section
34
35    } // end outer while
36
37 } // end Thread T1
```


5.4.1 Dekker's Algorithm

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

```
38
39 Thread T2:
40
41 void main()
42 {
43    while ( !done )
44    {
45       t2WantsToEnter = true;
46
47       while ( t1WantsToEnter )
48       {
49          if ( favoredThread == 1 )
50          {
51             t2WantsToEnter = false;
52             while ( favoredThread == 1 ); // busy wait
53             t2WantsToEnter = true;
54          } // end if
55
56       } // end while
```


5.4.1 Dekker's Algorithm

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

```
57
58       // critical section code
59
60       favoredThread = 1;
61       t2WantsToEnter = false;
62
63       // code outside critical section
64
65    } // end outer while
66
67 } // end Thread T2
```


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:

```
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)

```
1  System:
2
3  int favoredThread = 1;
4  boolean t1WantsToEnter = false;
5  boolean t2WantsToEnter = false;
6
7  startThreads(); // initialize and launch both threads
```


5.4.2 Peterson's Algorithm

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

```
8
9  Thread T1:
10
11 void main()
12 {
13    while ( !done )
14    {
15       t1WantsToEnter = true;
16       favoredThread = 2;
17
18       while ( t2WantsToEnter && favoredThread == 2 );
19
20       // critical section code
21
22       t1WantsToEnter = false;
23
24       // code outside critical section
25
26    } // end while
27
28 } // end Thread T1
29
```


5.4.2 Peterson's Algorithm

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

```
30  Thread T2:
31
32  void main()
33  {
34     while ( !done )
35     {
36        t2WantsToEnter = true;
37        favoredThread = 1;
38
39        while ( t1WantsToEnter && favoredThread == 1 );
40
41        // critical section code
42
43        t2WantsToEnter = false;
44
45        // code outside critical section
46
47     } // end while
48
49  } // end Thread T2
```


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. (1 of 3)

```
1  System:
2
3  // array that records which threads are taking a ticket
4  boolean choosing[n];
5
6  // value of the ticket for each thread initialized to 0
7  int ticket[n];
8
9  startThreads(); // initialize and launch all threads
10
```


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

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

```
11  Thread T_x:
12
13  void main()
14  {
15     x = threadNumber(); // store current thread number
16
17     while ( !done )
18     {
19        // take a ticket
20        choosing[x] = true; // begin ticket selection process
21        ticket[x] = maxValue( ticket ) + 1;
22        choosing[x] = false; // end ticket selection process
23
24        // wait for number to be called by comparing current
25        // ticket  value to other thread's ticket value
26        for ( int i = 0; i < n; i++)
27        {
28           if ( i == x )
29           {
30              continue; // no need to check own ticket
31           } // end if
32
```


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

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

```
33           // busy wait while thread[i] is choosing
34           while ( choosing[i] != false );
35
36           // busy wait until current ticket value is lowest
37           while ( ticket[i] != 0 && ticket[i] < ticket[x] );
38
39           // tie-breaker code favors smaller thread number
40           if ( ticket[i] == ticket[x] && i < x )
41
42              // loop until thread[i] leaves its critical section
43              while ( ticket[i] != 0 ); // busy wait
44        } // end for
45
46        // critical section code
47
48        ticket[x] = 0; // exitMutualExclusion
49
50        // code outside critical section
51
52     } // end while
53
54  } // end Thread TX
```