Computer Science & Engineering Department
I. I. T. Kharagpur

Operating System: CS33007

3rd Year CSE: 5th Semester (*Autumn 2006 - 2007*)
Lecture X (Mutual Exclusion of CS)

Goutam Biswas *Date:* 28th-29th August, 2006

1 Algorithmic Solution

Mutual exclusion of access to critical sections can also be achieved by algorithmic means without any support from the architecture. We present here two algorithms one by Dekker (presented by Dijkstra) and the other by Peterson. Both works for two processes. Then we go for generalization of these algorithms for more than two processes.

The basic requirements for the solution:

1. *Correctness (safety)*: control should not be in the CS of more than two processes or threads.
2. *Bounded wait (fair)*: no process should wait for indefinite amount of time.
3. *Progress*: if no process is in the critical section, a requesting process should not be stopped to enter its CS.

1.1 Several Attempts

Consider the first attempt to the solution. Let the processes be P_1 and P_2 and there is a shared variable `turn` initialized to 1. Consider the following pseudo code:

```
int turn = 1 ;          // shared data

/* Process 1 */

while(1){
     /* non-critical section 11 */
     while(turn == 2);  // Busy wait
     /* Critical section of code */
     turn = 2 ;
     /* non-critical section 12 */
}
/***************************************/

/* Process 2 */

while(1){
     /* non-critical section 21 */
     while(turn == 1);  // Busy wait
     /* Critical section of code */
     turn = 1 ;
     /* non-critical section 22 */
}
```

The shared variable `turn` indicates which process can enter its critical section and the *correctness* is satisfied. It also does not violate the *bounded wait*. But consider the following situation:

1. P_1 has a long non-critical section (12) where it may even be blocked.
2. P_1 comes out of its CS and and sets `turn = 2`.
3. P_2 enters the critical section, comes out of it, sets `turn = 1`, goes back into the loop, but cannot enter the CS even though P_1 is not trying to enter its CS

The condition of *progress* is violated. So the solution is not acceptable.

In the 2nd attempt let us try to solve the problem by introducing two shared variables in the following way:

```
int c1 = 0, c2 = 0 ; // shared data

/* Process 1 */

while(1){
     while(c2) ;     // Busy wait
     c1 = 1 ;
     /* Critical section */
     c1 = 0 ;
     /* Non-critical section 1 */
}

/* Process 2 */

while(1){
     while(c1) ;     // Busy wait
     c2 = 1 ;
     /* Critical section */
     c2 = 0 ;
     /* Non-critical section 2 */
}
```

This is certainly incorrect as both the processes can enter the CS together under the following condition.

1. Both $c1$ and $c2$ are 0.
2. P_1 executes `while` and is preempted.
3. P_2 executes `while` and is preempted.

4. Both enters the critical section.

In the 3rd attempt the claim to entry in the CS is established first.

```
int c1 = 0, c2 = 0 ; // shared data

/* Process 1 */

while(1){
     c1 = 1 ;
     while(c2) ;     // Busy wait
     /* Critical section */
     c1 = 0 ;
     /* Non-critical section 1 */
}

/* Process 2 */

while(1){
     c2 = 1 ;
     while(c1) ;     // Busy wait
     /* Critical section */
     c2 = 0 ;
     /* Non-critical section 2 */
}
```

This code may lead to dead-lock under the following situation:

1. P_1 executes `c1 = 1` and is preempted.
2. P_2 executes `c2 = 2`.
3. Both the processes will wait in their `while`.

What about the following code?

```
int c1 = 0, c2 = 0 ; // shared data

/* Process 1 */

while(1){
L:   while(c2) ;     // Busy wait
     c1 = 1 ;
     if(c2){
        c1 = 0 ;
        goto L ;
     }
     /* Critical section */
     c1 = 0 ;
     /* Non-critical section 1 */
}

/* Process 2 */

while(1){
L:   while(c1) ;     // Busy wait
     c2 = 1 ;
     if(c1){
        c2 = 0 ;
        goto L ;
     }
     /* Critical section */
     c2 = 0 ;
     /* Non-critical section 2 */
}
```

The mutual exclusion is guaranteed as the control can enter a CS when either (`c1 == 1` and `c1 == 0`) or (`c1 == 1` and `c1 == 0`). But then there is a possibility of *livelock*, both are trying to allow the other process to enter the CS.

1.2 Dekker's Algorithm

It uses both `turn`, `c1` and `c2` as shared variables:

```
\begin{verbatim}
int c1 = 0, c2 = 0, turn = 1 ;     // shared data

/* Process 1 */

while(1){
     /* non-critical section */
     c1 = 1 ;
     while(c2)
        if(turn == 2) {
                c1 = 0 ;
                while(turn == 2) ;  // Busy wait
                c1 = 1 ;
        }
     /* critical section */
     turn = 2 ;
     c1 = 0 ;
     /* non-critical section */
}

/* Process 2 */

while(1){
     /* non-critical section */
     c2 = 1 ;
     while(c1)
        if(turn == 1) {
                c2 = 0 ;
                while(turn == 1) ;  // Busy wait
                c2 = 1 ;
        }
      /* critical section */
      turn = 1 ;
      c2 = 0 ;
     /* non-critical section */
}
```

The variables `c1, c2` are request/status flags for the two processes P_1 and P_2. They register their request to enter the CS by making them 1.

The single variable `turn` can have one of two values and that changes only when a process comes out of its CS. This variable plays its role only when both processes have made the corresponding request flags 1 - process i enters its CS if `turn` is i. The value of `turn` is changed to $3 - i$ after P_i come out of the CS and the other process P_{3-i} gets the priority.

If P_1 want to enter the CS, it sets the request flag `c1 = 1`;

- if there is no request from P_2 i.e. `c2` is 0, `while(c2)` is false, and P_1 enters its CS. P_2 cannot enter its CS as `c1` is 1.
- if there is a request from P_2 i.e. `c2` is 1, `while(c2)` is true. The value of `turn` is tested. If it is 1 (in favour of P_1), P_1 enters the critical section. This itself stops P_2 to enter its CS. P_1 sets `turn` to 2 after coming out of its CS and allows P_2 P_2 to enter.

Following is an implementation of Dekker's algorithm for producer-consumer problem on pthreads.

```
/*****************************************************************
 * The producer consumer problem using a thread
 * CS using Dekker's algorithm
 * cc -Wall -c queue.c
 * cc -Wall -lpthread dekkerProdCons.c queue.o
 * ***************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include "queue.h"

queue q ;
int c1, c2, turn ;

void * thread1(void *) ;
void * thread2(void *) ;
void producer(){
     int count = 1 ;
     while(1) {
          int data, added = 1, err ;

          if(added) {
             data = rand() ;
             added = 0 ;
          }
// Dekker algorithm
          c1 = 1 ;
          while(c2){
               if(turn == 2){
                  c1 = 0 ;
                  while(turn == 2) ;
                  c1 = 1 ;
               }
          }
//
          err = addQ(&q, data) ;
// DA
          turn = 2 ;
          c1 = 0 ;
//
          if(err == OK) {
               added = 1 ;
               printf("Produced Data %d: %d\n", count++, data) ;
          }
     }

}

void consumer() {
     int count = 1 ;
     while(1) {
          int data, err ;

// Dekker algorithm
          c2 = 1 ;
          while(c1){
               if(turn == 1){
                  c2 = 0 ;
                  while(turn == 1) ;
                  c2 = 1 ;
               }
          }
//
          err = frontQ(&q, &data) ;
          if(err == OK) err = deleteQ(&q) ;
// DA
          turn = 1 ;
          c2 = 0 ;
//
          if(err == OK)
             printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
     }
}

int main() {
     pthread_t thID1, thID2 ;

     initQ(&q) ;
     c1 = 0 ; c2 = 0; turn = 1; // Dekker algorithm variables
     pthread_create(&thID1, NULL, thread1, NULL) ;
     pthread_create(&thID2, NULL, thread2, NULL) ;
     pthread_join(thID2, NULL) ;
     pthread_join(thID1, NULL) ;
     return 0;
}

void *thread1(void *vp) {
     producer() ;
     return NULL ;
}

void *thread2(void *vp) {
     consumer() ;
     return NULL ;
}
```

1.3 Peterson's Algorithm

A simpler and elegant algorithm was proposed by Peterson to solve two process mutual exclusion problem.

```
int c1 = 0, c2 = 0, turn ; // Shared data

/* Process 1 */

while(1){
     /* non-CS  */
     c1 = 1 ;
     turn = 1 ;
     while(c2 && turn == 1) ;
     /* Critical section */
     c1 = 0 ;
     /* non-CS  */
}

/* Process 2 */

while(1){
     /* non-CS  */
     c2 = 1 ;
     turn = 2 ;
     while(c1 && turn == 2) ;
     /* Critical section */
     c2 = 0 ;
     /* non-CS  */
}
```

The variables `c1, c2` are request/status flags for the two processes P_1 and P_2. They register their request to enter the CS by making them 1.

The single variable `turn` can have one of two values 1 or 2. Its role is different from Dekker's algorithm. The process P_2 sets the request flag `c2` to 1 for entering the critical section, it also assigns 2 to `turn`.

- If there is no request from P_1 so far, P_2 enters the critical section. P_1 cannot enter it as long as `c2` is 1. P_2 after coming out of its CS makes `c2` 0 and P_1 may enter if necessary.
- Even if both `c1` and `c2` are 1, the value of `turn` can be either 1 or 2. Depending on this value either P_1 or P_2 will busy wait.

Following is an implementation of Peterson's algorithm for producer-consumer problem on pthreads.

```
/*************************************************************
 * The producer consumer problem using a thread
 * CS using Peterson's algorithm
 * cc -Wall -c queue.c
 * cc -Wall -lpthread petersonProdCons.c queue.o
 * ************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include "queue.h"

queue q ;
int c1, c2, turn ;
```

```
void * thread1(void *) ;
void * thread2(void *) ;

void producer(){
     int count = 1 ;
     while(1) {
          int data, added = 1, err ;

          if(added) {
                  data = rand() ;
                  added = 0 ;
          }
// Peterson' algorithm
          c1 = 1 ;
          turn = 1 ;
          while(c2 && turn == 1) ;
//
          err = addQ(&q, data) ;
// PA
          c1 = 0 ;
//
          if(err == OK) {
               added = 1 ;
               printf("Produced Data %d: %d\n", count++, data) ;
          }
     }

}

void consumer() {
     int count = 1 ;
     while(1) {
          int data, err ;

// Peterson's algorithm
          c2 = 1 ;
          turn = 2 ;
          while(c1 && turn == 2) ;
//
          err = frontQ(&q, &data) ;
          if(err == OK) err = deleteQ(&q) ;
// DA
          c2 = 0 ;
//
          if(err == OK)
             printf("\t\t\tConsumed Data %d: %d\n", count++, data) ;
     }
}

int main() {
     pthread_t thID1, thID2 ;

     initQ(&q) ;
     c1 = 0 ; c2 = 0;  // Peterson algorithm variables
     pthread_create(&thID1, NULL, thread1, NULL) ;
     pthread_create(&thID2, NULL, thread2, NULL) ;
     pthread_join(thID2, NULL) ;
     pthread_join(thID1, NULL) ;
     return 0;
}

void *thread1(void *vp) {
     producer() ;
     return NULL ;
}

void *thread2(void *vp) {
     consumer() ;
     return NULL ;
}
```

1.4 Bakery Algorithm

The mutual exclusion of CSs of n processes can be achieved by the following algorithm of L. Lamport.

When a processes wishes to enter its CS, it takes an integer *token*. The value of the token is larger than the tokens issued earlier. Process enter the CS in order of the *token*. But getting token is a concurrent activity and two processes may get the same token.

There is an array `choosing[]` of boolean flags, `choosing[i]` indicates that the process P_i is trying to get a token. The value of the token for the process P_i is in `number[i]`. It is 0 if the process has not taken any token after it has come out of its CS.

```
const int n = ... ;
int choosing[n], number[n] ;

for(i=0; i<n; ++i){
   choosing[i] = 0 ; number[i] = 0 ;
}

/******* Process Pj *******/

while(1){
     /* non-CS */
     choosing[j] = 1 ;
```

```
    number[j] = max(number, n) + 1 ;
    choosing[j] = 0 ;

    for(i=0; i<n; ++i) {
       while(choosing[i]) ;  // busy wait
       while(number[i] != 0 && (number[i],i) < (number[j], j)) ; // busy wait
    }
    /*  CS  */
    number[j] = 0 ;
    /* non-CS */
}
```

References

[1]