Operating Systems
Lecture #4

THE CRITICAL SECTION PROBLEM

1. MOTIVATION OF CRITICAL SECTION PROBLEM

In order to explain the motivation of critical section in concurrent processes, let us illustrate simple example that is representative of operating systems.

THE PRODUCER/CONSUMER PROBLEM

1) Problem description

Producer/consumer processes are common in operating systems A producer process produces information that is consumed by a consumer process (A compiler produce assembly code, which is consumed by an assembler).

To allow producer and consumer processes to run concurrently, we must create a pool of buffers that can be filled by the producer and consumed by the consumer. The bounded buffer producer/consumer problem assumes that there is a fixed number, n, of buffers (the consumer wait all the buffers are empty and the producer must wait if all the buffers are full).

The producer and consumer must be synchronized so that the consumer does not try to consume items which have not yet been produced.

2) Solution to the bounded buffer problem

The shared variables are:

```
TYPE item = ... ;

 VAR buffer: ARRAY[0..n-1] OF item;
  /* Circular array with two logical pointers in, out */
  counter: INTEGER;  /* Initialize to 0 */
  in, out: 0..n-l;     /* in points to the next free buffer, our points to the first full
                          buffer. The pool empty when in=out while full when in+l
                          MOD n = out.  Initialize to the value  0 */
```

```
PARBEGIN
  PRODUCER: BEGIN
    REPEAT
        ....
    produce an item in nextp  /* nextp is a local variable in which the new item to
                                 be produces is stored */
        ....
    WHILE counter = n DO skip;  /* tests the condition repeatively until it
                                   becomes false */
      buffer[in] := nextp;
      in := (in + 1) MOD n;
      counter := counter + 1;
    UNTIL false;
  END PRODUCER;

  CONSUMER BEGIN
    REPEAT
      WHILE counter = 0 DO skip;
      nextc:  buffer[out];
      /* local variable nextc in which the item to be consumed */
      out := (out + 1) MOD n;
      counter := counter -1;
        ....
      consume the item in nextc
        ....
    UNTIL false;
  END CONSUMER
PAREND;
```

3) Observation

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.

Suppose that the value of counter is currently 5 and that the procedure and consumer processes execute the statements "counter := counter + 1" and "counter := counter -1" concurrently. We define that Ti is time instance in CPU where i = 0 to n.

```
T0: producer EXECUTE registerl := counter          {registerl = 5}
T1: producer EXECUTE registerl := registerl + 1    {registerl = 6}
T2: consumer EXECUTE register2 := counter          {register2 = 5}
T3: consumer EXECUTE register2 := register2 - 1    {register2 = 4}
T4: producer EXECUTE counter := registerl          {counter = 6}
T5: consumer EXECUTE counter := register2          {counter = 4}
```

Notice that we have at the incorrect state "counter = 4" recording that there are four full buffers when in fact there are five full buffers. If we reverse the order of the statement at T4 and T5 we would arrive at the incorrect state "counter = 6".

4) Conclusion

We may arrive at this incorrect state because we allowed both processes to manipulate the variable "counter" concurrently. In order to solve this problem, we need to ensure that only one process at a time may be manipulating the variable "counter". This observation leads us to critical section problem.

2. PROBLEM DEFINITION

Consider a system consisting of n cooperating processes {P1, P2 ,..., Pn}.

1) Definition of critical section:

Each process has a segment of code, called a critical section, in which the process may be reading common variables, updating a table, writing a file, and so on.

2) Definition of mutual exclusion:

When one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus the execution of critical section by the processes is mutually exclusive in time.

3) Three requirements of the mutual exclusion

a) Mutual Exclusion.

If process Pi is execution in its critical section then no other process can be executing in its critical section.

b) Progress.

If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then only those processes that are not executing in their remainder section can participate in the decision as to who will enter the critical section next; and this selection cannot be postponed indefinitely.

c) Bounded Waiting.

There must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

3. TWO-PROCESS SOFTWARE SOLUTIONS

We trace the initial attempts made in trying to develop algorithms for ensuring mutual exclusion. General structure of the problem is:

```
BEGIN
  common variable declarations;
  PARBEGIN
    P0;
    P1;
  PAREND;
END.
```

1) Algorithm 1

a) Descriptions

* Common integer variable turn initialize to 0 (or 1).
* If turn = i, then process Pi is allowed to execute in its critical section.
  If turn = j, then process Pj is allowed to execute in its critical section.

```
P0:
 REPEAT
   WHILE turn <> i DO skip;
       Critical section
   turn := j;
       remainder section
 UNTIL false;
```

```
P1:
 REPEAT
   WHILE turn <> j DO skip;
       Critical section
   turn := i;
       remainder section
 UNTIL false;
```

b) Discussion

i) Mutual exclusion is guaranteed. Only one process at a time can be its critical section.

ii) Violate progress requirement:
It requires strict alternation of *processes* in the execution of the critical section. Suppose turn = 0 and P0 is in remainder section. At that moment, P1 wants to enter the critical section but P1 can not do so even though P0 is in its remainder section.

iii) Violate bounded waiting requirement:
If turn = i and Pi do not want to enter the critical section then Pj may wait forever.

iv) The problem is that it fails to remember the state of each process, but remembers only which process is allowed to enter its critical section.

2) Algorithm 2

Variable turn with the following array.

VAR flag: ARRAY [0..1] OF boolean; /* initialize to false */
If flag[i] is true, then process Pi is executing in its critical section.
If flag[j] is true, then process Pj is executing in its critical section.

```
P0:
  REPEAT
    WHILE flag[j] DO skip;¡
      flag[i] := true;
        critical section  ¡
      flag[i] := false;
        remainder section
  UNTIL false;
```

```
P1:
  REPEAT
    WHILE flag[i] DO skip;
      flag[j] := true;
        critical section
      flag[j] := false;
        remainder section
  UNTIL false;
```

a) Discussion

i) Violating the mutual-exclusion requirement.
T0: P0 enters the while statement and find flag[1] = false.
T1: P1 enters the while statement and finds flag[0] = false.
T2: P1 set flag[l] = true and enters the critical section.
T3: P0 sets flag[0] = true and enters the critical section.

3) Algorithm 3

a) Description

The problem with algorithm 2 is that process Pi made a decision concerning the state of Pj before Pj had the opportunity to change the state of the variable flag[j]. This time, the setting of flag[i] = true indicates only that Pi wants to enter the critical section.

```
REPEAT
  flag[i] := true;
  WHILE flag[j] DO skip: ¡
     critical section
  flag[i] := false;
     remainder section
UNTIL false;
```

```
REPEAT
   flag[j] := true;
WHILE flag[i] DO skip;
     critical section
   flag[j] := false;
     remainder section
UNTIL false;
```

b) Discussion

* Guarantee mutual exclusion.
* Violate the progress requirement.

T0: P0 sets flag[0] = true.
T1: P1 sets flag[l] = true.
Now P0 and P1 are looping forever in their respective WHILE statements.

4) Algorithm 4 (Peterson Algorithm)

a) Processes share two variables in common:

```
VAR flag: ARRAY [0..l] OF boolean;
     turn: 0..1;
```

Initially flag[0] = flag[l] = false and the value of turn is immaterial (either 0 or 1).

```
P0:                                      P1:
 REPEAT                                   REPEAT
    flag[i] := true;   ¡                      flag[j] := true;
    turn := j;                                        turn := i;
    WHILE (flag[j] AND turn=j) DO skip        WHILE (flag[i] AND turn = i) DO skip;
          critical section                          critical section
       flag[i] := false;                         flag[j] := false;
          remainder section                         remainder section
UNTIL false;                               UNTIL false;
```

b) Discussion

i) To prove property of mutual exclusion

We note that each Pi enters its critical section only if either section only if either flag[j] = false or turn = i. Also note that if both processes could be executing in their critical sections at the same time then flag[0] = flag[l] = true. These two observations imply that P0 and P1 could not have successfully executed their WHILE statement at about the same time, since the value of turn can be either 0 or 1, but not both. Hence, one of the processes, say Pj, must have successfully executed the WHILE statement, while Pi had to at least execute one additional statement "turn = j". However, since at that point in time flag[j] = true, and turn = i, and this condition will persist as long as Pj is in its critical section, the result follows: mutual exclusion is preserved.

ii) To prove properties Progress and Bounded waiting, a process Pi can be prevented from entering the critical section only if it stuck in the WHILE loop with the condition flag[j] = true and turn = j; this is the only loop. If Pj is not interested in entering the critical section, then flag[j] = false and Pi can enter its critical section. If Pj has set flag[j] = false and is also executing in its WHILE statement,

then either turn = i or turn = j. If turn = i, then Pi will enter the critical section If turn = j, then Pj will enter the critical section. However, once Pj exits its critical section, it will reset flag[j] to false allowing Pi to enter its critical section. If Pi does not change the value of the variable turn while executing the while statement, Pi will enter the critical section (Progress) after at most one entry by Pj (bounded-wailing).

5) Dekker's Algorithm

The first known correct software solution processes, P0 and P1, share the following variables:

```
VAR  flag: ARRAY [0..1] OF boolean;  /* Initially false */
     turn: 0..l;
```

The program below is for process Pi (i = 0 or 1) with process Pj (j = 1 or 0) being the other one.

```
REPEAT

    flag[i] := true;
    WHILE flag[j] DO
      IF turn = j THEN
         BEGIN
           flag[i] := false;
           WHILE turn= j DO skip;
              flag[i] := true;
         END;
       ...
       critical section
       ...
      turn := j;
      flag[i] := false;
       ...
       remainder section
       ...
UNTIL false;
```

4. HARDWARE SOLUTIONS

i) INTRODUCTION

The critical section problem could be simply solved if we could disallow interrupts to to occur while a shared variable is being modified. Many machines, thus, provide special hardware instructions that allow one to either test and modify the contents of two

words, or to Swap the contents of two words, in one instruction cycle.

Let us abstract the main concepts behind these types of instructions by defining the Test-and-Set instruction as follows:

```
FUNCTION Test-and-Set (VAR target : boolean): boolean;
  BEGIN
    Test-and-set := target;
    target := true;
  END;
```

and the Swap instruction as follows:

```
PROCEDURE Swap (VAR a, b: boolean);
  VAR temp : boolean;
  BEGIN
    temp := a;
    a := b;
    b := temp;
  END;
```

These instructions are executed atomically (one instruction cycle).
If the machine supports the Test-and-Set instruction, then mutual exclusion can be implemented by declaring a boolean variable lock, initialize to false.

```
REPEAT
   WHILE Test-and-Set(lock) DO skip;
      critical section
   lock := false;
      remainder section
UNTIL false;
```

If the machine support the Swap instruction, then mutual exclusion can provided in a similar manner. A global boolean variable lock is declared which is initialize to false. In addition, each process also has a local boolean variable key.

```
REPEAT

   key := true;
   REPEAT
      Swap (lock, key);
   UNTIL key = false;
```

```
     critical section
   lock := false;
     remainder section
UNTIL false;
```

Test_and_Set
VAR Active:Boolean;

```
PRODUCER Process_One
   VAR One_Cannot_Enter: Boolean
   BEGIN
      One_Cannot_Enter := True;
      WHILE One_Cannot_Enter DO
        Test_And_Set(One_Cannot_Enter, Active);
             .........
        Critical Section;
             .........
        Active := False;
             .........
        Remainder Section;
      END_WHILE
   END
```

The algorithms presented above do not satisfy the bounded-waiting requirement. To do so additional variables must be used. Below, we present an algorithm that uses the Test-and-Set instruction, and which satisfies all the required critical section requirements.

The common data structures are:

```
  VAR waiting: array[l..n-1] OF boolean
      lock: BOOLEAN
```

These data structures are initialize to false.

The structure of process Pi is:

```
   VAR j: 0..n-1;
      key: BOOLEAN;

   REPEAT
      waiting[i] := true:
      key := true;
      WHILE waiting[i] AND key DO key := Test-and-Set(lock);
         waiting[i] := false;
```

```
        critical section
        j := i+l MOD n;
    WHILE (j <> i) AND (NOT waiting[j]) DO j := j + 1 MOD n;
       IF j = i THEN lock := false
          ELSE waiting[j] := false;
        remainder section
  UNTIL false;
```

To prove that the mutual-exclusion requirement is met, we note that process Pi can enter its critical section only if either waiting[i] = false or key = false. Key can become false only by executing the Test-and-Set. The first process to execute the Test-and-Set will find key = false; all others must wait. Waiting[i] can become false only if another process leaves its critical section; only one waiting[i] is set true, maintaining the mutual-exclusion requirement.

To prove the progress requirement, we note that the arguments presented above for mutual exclusion also apply here, since a process exiting the critical section either sets lock to false, or waiting[i] = false.
Both allow a trying process to enter its critical section.

To prove bounded-waiting. we note that when a process leaves its critical section, it scans the array waiting in the cyclic ordering (i+l. i+2,..., n-1, 0, ..., i-1). It designates the first process in this ordering which is in its entry section (waiting[j] = true) as the next one to enter its critical section. Any process waiting to enter its critical section will thus do so within n-l turns.