Operating Systems
Lecture #6

CONCURRENT PROGRAMMING

1. MOTIVATION

Operating systems are now almost always written in a system implementation language or a higher-level language. This feature improves their implementation maintenance, and portability.

1) Advantages of higher-level programming language over assembly language.

*** Simpler and easier to test. to modify, and to transfer from are machine to another.**
*** The programmer spends less time developing and testing programs.**

2) Disadvantage of higher-level language.

Inefficient code generation (this problem can be resolved by the use of various compiler optimization techniques).

In this lecture we are mainly concerned with higher-level languages that are to be used in writing concurrent programs. Such languages must provide facilities for modularization and synchronization.

2. MODULARIZATION

Modularization is the term used to describe the partitioning of a single large program into a set of smaller modules.

1) PROCESSES

A process is a fundamental building block in the design of an operating system.

In the most case processes can sham all variables. In this case, to guard against .time -dependent errors, programmers must build their own synchronization schemes. The compiler cannot aid the programmer in this task. In the most restrictive case, processes do not share any variables. Communication and synchronization can be accomplished by a message-passing facility or by parameter passing separate procedures.. The compiler can ensure at compile time that no sharing occurs, hence there will be no time-dependent errors caused by sharing variables.

A process thus consists of some local data, and a sequential program that can operate

on the data. The local data can only be access by the sequential program encapsulated within the same process. That is, one process cannot directly access the local data of another process. If processes can share global data, this data must be either defined in a common area, or encapsulated in a procedure or abstract data type.

2) PROCEDURES

Procedure axe the elementary program structure for information hiding.

3) ABSTRACT DATA TYPES

a) Introduction

A procedure provide us with a limited mechanism for information hiding. To hide the method of defining data completely, we must resort to a more complicated set of mechanisms. We need a facility that allows a programmer to create a class of abstract objects that were not explicitly anticipated in the design of the programming language.

b) Definition of abstract data type.

An abstract data type is characterized by a set of programmer-defined operation. The representation of an abstract data type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodes of procedures or functions that implement operations on the type.

Example of an abstract data type is a stack which is a programer-defined data type. not language-defined data type. A stack should only be accessed by push, pop. empty, and top operations; elements of the stack should never be directly accessed.

We refer to a program module that gives the representation of an abstract type as a CLASS. The syntax of a CLASS is:

```
TYPE class name = CLASS
    variable declarations

    PROCEDURE P1 (formal parameters):
       BEGIN ... END;

    PROCEDURE P2 (...);
       BEGIN ... END:

    PROCEDURE Pn (...);
       BEGIN ... END;

    BEGIN
       initialization code
    END
```

A procedure defined within a CLASS can only access the variables declared locally within the class and the formal parameters. Similarly, the local variables of a class can only be accessed by the local procedures.

A module exports the name and types of its operators, but does not export the representation of the type it defines. Direct access to the representation of the type can only be done within the class in which the new type is defined.

A CLASS need not export all of its procedures. To distinguish between procedures that are exported and those that am not, we use the notation:

```
PROCEDURE ENTRY P (...);
```

Only entry procedure can be invoked from outside the CLASS.

c) Example of abstract data type

Suppose that we want to define a CLASS for distributing. fixed-size storage such as pages in memory or on a block a disk. The information about the free storage is kept in a bit vector, which is encapsulated within the frame CLASS described below.

```
TYPE frame = CLASS
    VAR free:ARRAY [l...n] OF boolean;

    PROCEDURE ENTRY acquire (VAR index: integer);
      BEGIN
        FOR index := 1 to n DO
          IF free[index] THEN
            BEGIN
              free[index] := false;
              exit;
            END;
        index := -1;
    END;

    PROCEDURE ENTRY release (index: integer);
      BEGIN
        free[index] := true;
      END;

BEGIN
  FOR index := 1 TO n DO
    free[index] := true;
END
```

An instance of the frames class can Do obtained by declaring:

```
VAR memory: frames;
```

A process needing to acquire a free page will execute:

```
memory.acquire (in);
```

If a free page is not available, the variable in will be set to -1. At this point, the process must wait or request page swapping. The process may return the allocates page by executing:

```
memory.release (in);
```

The advantage of the CLASS definition of frames is that we can easily change the implementation of frames without changing its use.

3. SYNCHRONIZATION

In the frames example. several processes may invoke the procedures acquire and release simultaneously. This situation may result in inconsistent data. To guard against this possibility, the language must also provide appropriate synchronization tools.

A language must provide the means to guard against time-dependent errors. Such errors can occur if several concurrent processes communicate with each other by the use: of either common variables or explicit messages. Obviously, we could disallow such communication entirely, thus eliminating time-dependent errors. This restriction, however, will drastically reduce the amount of concurrency allowed in an operating system. Clearly, this solution is not viable. Therefore. we will have to compromise. Instead of completely eliminating time-dependent errors, we will provide language constructs that will reduce considerably.

1) CRITICAL REGIONS

a) Motivation

The time dependent errors can be easily generated when semaphores are used to solve the critical section problem. To overcome this difficulty a new language construct, the critical region was introduced.

b) Definition and notation

A variable v of type T, which is to be shared among many processes, can be declared:

```
VAR v: SHARED T;
```

The variable v can be accessed only inside a region statement of the following form:

```
REGION v DO S;
```

This construct means that while statement S is being executed, no other process can access the variable v. Thus, if the two statements,

```
REGION v DO S1;
REGION v DO S2;
```

are executed concurrently in distinct sequential processes, the result will be equivalent to the sequential execution S1 followed by S2, or S2 followed by S1.

To illustrate this construct, consider the frames CLASS defined in abstract data type. Since mutual exclusion is required when accessing the array free, we need to declare it as a shared array.

```
VAR free: SHARED ARRAY [l..n] OF boolean;
```

The acquire procedure must be rewritten as follows;

```
PROCEDURE ENTRY acquire (MAR index: integer);
  BEGIN
    REGION free DO
      FOR index := 1 TO n DO
        IF free[index] THEN
          BEGIN
            free[index] := false;
            exit;
          END;
    index :=  1;
  END;
```

The critical-region construct guards against some simple errors associated with the semaphore solution to the critical section problem which may be made by a programmer.

c) Compiler implementation of the critical region construct.
For each declaration

```
VAR v: SHARED T;
```

the compiler generates a semaphore v-mutex initialized to 1. For each statement,

```
REGION v DO S;
```

the compiler generates the following code:

```
p(v-mutex);
S;
V(v-mutex);
```

Critical region may also be nested. In this case, however, deadlocks may result. Example of deadlock)

```
VAR x, y: SHARED T;
    PARBEGIN
        Q: REGION x DO REGION y DO S1;
        R: REGION y DO REGION x DO S2;
    PAREND;
```

2) CONDITIONAL CRITICAL REGIONS

The critical region construct can be effectively used to solve the critical section problem. It cannot, however, be used to solve some general synchronization problems. For this reason the conditional critical region was introduced.

The major difference between the critical region and the conditional critical region constructs is in the region statement, which now has the form:

```
REGION v WHEN B DO S;
```

where B is a boolean expression.
As before, regions referring to the same shared variable exclude each other in time. Now, however, when a process enters the critical section region. the boolean expression B is evaluated if the expression is true, statement S is executed. if it is false, the process relinquishes the mutual exclusion and is delayed until B becomes lame and no other process is in the region associated with v.

Example for bounded buffer problem)

```
VAR buffer: SHARED RECORD
      pool: ARRAY [0..n-l] OF item;
      count, in. out: integer;
    END;
```

The producer process inserts a new item nextp in buffer by executing

```
REGION buffer WHEN count < n DO
    BEGIN
      pool[in] := nextp;
      in := in + i MOD n;
      count := count + 1;
    END;
```

The counter process removes an item from the shared buffer and puts it in nextc by executing:

```
REGION buffer WHEN count > 0 DO
            BEGIN
              nextc := pool[out];
              out := out +1 MOD n;
              count := count - 1;
            END;
```

However, the CLASS concept alone cannot guarantee that such sequences will be observed.

*** A process might operate on the file without first gaining access permission to it;**
*** A process might never release the file once it has been granted access to it;**
*** A process might attempt to release a file that it never required;**
*** A process might request the same file twice;**

Not that we have now encountered difficulties that are similar in nature to those that motivated us to develop the critical region construct in the first place. Previously, we had to worry about the correct use of Semaphores. Now we have to worry about the correct use o higher-level programmer-defined operations, with which the comelier can no longer assist us.

4. MONITORS

1) MOTIVATION

In the preceding sections, we have described the critical-region mechanisms for process synchronization. We have also combined this mechanism with the CLASS concept to achieve a greater degree of protection. One unattractive feature of this combination was that every procedure had to provide its own synchronization explicitly. A desired to provide the appropriate synchronization automatically led C.A.R Hoare to the development of a new language construct a MONITOR.

2) DEFINITION

A monitor is a mechanism that allows the safe and effective sharing of abstract data types among several processes. The syntax of a monitor is identical to that of a class, except that the keyword CLASS is replaced by the keyword MONITOR. The main semantic difference is that the monitor ensures mutual exclusion; that is, only one process at a time can be achieve within the monitor. This properly is guaranteed by the monitor itself. Consequently. the programmer need not explicitly code this synchronization constraint.

We need additional mechanism with monitor for synchronization. These mechanisms are provided by the CONDITION construct. Programmers who need to write their own tailor-made synchronization scheme can define one or more variables of type CONDITION: VAR x, y: CONDITION;

The only operation that can be invoked on a CONDITION variable are wait and signal. Thus a condition variable can be viewed as an abstract data type that provides those two operations. The operation

```
x.wait;
```

means that the process invoking this operation is suspended until another process invokes

```
x.signal;
```

The x.signal operation resumes exactly one suspended process. If no process is suspended, then the signal operation has no effect; that is, die state of x is as if the operation was never executed. Contrast this with the V operation, which always affects the state of the semaphore.

Now suppose that when die x.signal operation is invoked by a process p, there is a suspended process Q associated with CONDITION x. Clearly, if the suspended process Q is allowed to resume its execution. the signaling process p must wait. Otherwise, both p and Q will be active simultaneously within the monitor. When both processes can conceptually continue with their execution, p wait until Q either the monitor, or waits for another CONDITION.

3) SIMULATION OF THE SEMAPHORE

Let us illustrate these concepts by writing a monitor that simulates a binary semaphore.

```
TYPE semaphore = MONITOR
VAR busy: boolean;
   nonbusy: CONDITION;

   PROCEDURE ENTRY p;
     BEGIN
        IF busy THEN nonbusy.wait;
        busy := true;
     END;

   PROCEDURE ENTRY v;
     BEGIN
        busy := false;
        nonbusy.signal;
     END;
```

```
BEGIN
   busy := false;
END.
```

The boolean variable busy indicates the state of the semaphore. When the first P operation is executed, the state of the semaphore is set to busy; if any additional P operations am attempted, these processes must wait until a V operation occurs. Notice that the V operation always sets busy to false. If there is a waiting process, it immediately resets busy to true. If no process is waiting. busy remains false.

4) IMPLEMENTATION OF THE MONITOR MECHANISM.

We will now consider a possible implementation of the monitor mechanism. For each monitor, a semaphore mutex (initialized to 1) is provided. P(mutex) must be executed before entering the monitor. while V(mutex) must be executed after leaving the monitor.

Since a signaling process must wait until the resumed process either leaves or waits, an additional semaphore, next, is introduced, initialized to 0, on which the signaling processes may suspended themselves. An integer variable next_count will also be provided to count the number of processes suspended on next.

```
P(mutex);
...
body of F;
...
IF-next_count > 0 THEN V(next)
                  ELSE V(mutex);
```

Mutual exclusion within a monitor is ensured.

5) IMPLEMENTATION OF CONDITION VARIABLES.

For each CONDITION x we introduce a semaphore x.sem and an integer variable x.count, both initialized to 0. The operation x.wait can now be implemented as

```
x-count :- x-count + 1;
IF next-count > 0 THEN V(next)
                  ELSE V(mutex);
P(x-sem);
x-count := x count - 1;
```

The operation x.signal can be implemented as

```
IF x-count > 0 THEN
  BEGIN
    next_count := next_count + 1;
    V(x-sem);
    P(next);
    next_count := next_count - 1;
  END;
```

6) NESTED MONITOR CALLS PROBLEM.

Note that if a monitor M1 calls another monitor M2 the mutual exclusion in MI is not released while execution proceeds in M2. This fact has two consequences:

i) Any process calling MI will be blocked outside M1 on mutex during this time period.
ii) If the process enters a condition queue in M2, a deadlock may occur.

7) PROCESS RESUMPTION ORDER

That is, if several processes are suspended on condition x. and an x.signal operation is executed. which process is resumed next? One simple scheme is to use a first-come -first-served scheme. Thus the process waiting longest is resumed first. There arc, however. many circumstances when such a simple scheduling scheme is not adequate.

For this reason. conditional wait was introduced. It has the form x. wail(c); where c is an integer expression that is evaluate when the wait operation is executed. The value of c, which is called a priority number. is then stored with the name of the process that is suspended. When x.signal is executed, the process with the smallest associated priority number is resumed next.

Example for Shortest-Job-First scheduling

```
TYPE SJN = MONITOR

  VAR busy: boolean;
    x: CONDITION;

  PROCEDURE ENTRY acquire (time: integer);
    BEGIN
      IF busy THEN x.wait (time);
      busy := true;
    END;
```

```
PROCEDURE ENTRY release;
   BEGIN
     busy := false;
     x.signal;
   END;

BEGIN
   busy := false;
END.
```