Operating Systems
Lecture #5

SEMAPHORES

1) MOTIVATION

The solutions to the mutual exclusion problem presented in the list lecture are not easy to generalize to more complex problems. To overcome this difficulty, a new synchronization tool, semaphore, was introduced by Dijkstra in 1965.

2) DEFINITION

A semaphore S is an integer variable chat. apart from initialization, can be accessed only through two standard atomic operations: P and V.

```
P(S): WHILE S <= 0 skip;
      S := S - 1;

V(S): S:= S + 1;
```

Note: When one process modifies the semaphore value. no other process can simultaneously modify that same semaphore value. In the case of the P(S), the testing of the integer value of S (S <= 0) and its possible modification (S := S-1) must also be executed without interruption.

3) USAGE

a) MUTUAL EXCLUSION: semaphore deals with n-process critical section problem.

* Each process i is organized as follows:

```
REPEAT
   P(mutex);  /* common semaphore mutex initialized to 1 */
      ...
   critical section
      ...
  V(mutex);
      ...
   remainder section
      ...
UNTIL false;
```

b) SYNCHRONIZATION: semaphore deals with precedence relationship between processes.

*** Example - 1)**

Two process running concurrently. P1 with statement S1, P2 with statement S2.

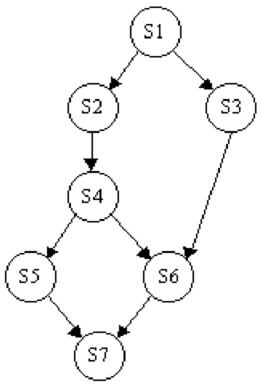
```
P1:    S1;
       V(synch);  /* common semaphore synch initially 0 */
P2:    P(synch);
       S2;
```

Note: P2 be executed only after SI has completed.

*** Example - 2)**

```
VAR a, b, c, d, e, f, g: semaphores; /* initial value of all is 0 */

BEGIN
  PARBEGIN
      BEGIN S1; V(a); V(b); END;
      BEGIN P(a); S2; S4; V(c); V(d); END;
      BEGIN P(b); S3; V(e); END;
      BEGIN P(c); S5; V(f); END;
      BEGIN P(d); P(e); S6; V(g); END;
      BEGIN P(f); P(g); S7; END;
  PAREND;
END;
```



4) IMPLEMENTATION

a) Issue:

The major disadvantage of the software solution for mutual exclusion and the semaphore definition given above is busy waiting which wastes CPU cycles.

b) Solution

Need to modify the P, V semaphore operations. When a process executes the P operation and finds that the semaphore value is not positive, it must wait. However, rather than busy-waiting, the process can BLOCK itself. The block operation places a process into a waiting state. If then transfers control to the cpu scheduler, which selects another process to execute from the ready queue.

A process which is blocked, waiting on a semaphore S, should be restarted by the execution of a V operation by some other process. The process is restarted by a WAKEUP operation, which changes the state of the process from blocked to ready and put it into the ready queue.

```
TYPE semaphore = RECORD
                      value: INTEGER;
                      L: LIST OF process;
                   END_RECORD;

/* each semaphore has an integer value and list of processes */

P(S): S.value := S.value - 1;
      IF S.value < 0 THEN
         BEGIN
              add this process to S.L;
              block; /* suspends the process that invokes it */
         END;

V(S): S.value := S.value + 1;
      IF S.value <=  0 THEN
         BEGIN
              remove a process P from S.L;
              wakeup(P); /* resumes the execution of a blocked process P */
         END;
```

c) Notes:

* When a process must wait on a semaphore, it is added to the list of processes A V operation remove one process from the list of waiting process and awakens it.
* If the semaphore value is negative, its magnitude is the number of processes

waiting.

* We must guarantee that no two processes can execute P and V operations on the same semaphore at the same time.

* We have removed busy-waiting from the entry to the critical sections rather than completely eliminated busy waiting with this solution.

5) DEADLOCKS

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can only be caused by one of the waiting processes. The event in is the execution of a V operation. When such a state is reached, these processes are said to be deadlocked.

Example)

```
/* initially S and Q = 1 */
PO                  P1

P(S);               P(Q);
P(Q);               P(S);
 ...                 ...
V(S);               V(Q);
V(Q);               V(S);
```

* Suppose that P0 executes P(S) and then P1 executes P(Q). When P0 executes P(Q) it must wait until P1 executes V(Q). Similarly, when P1 executes P(S) it must wait until P0 executes V(S). Since these V operations can not be executed, P0 and P1 are deadlocked.

6) TIME DEPENDENT ERRORS

If processes can share variables in an arbitrary manner, then time dependent errors can occur. Following examples illustrate that time-dependent error can easily generated when semaphores are used to solve the critical section problem and synchronization problems.

a) Suppose that a process interchanges the operations on the semaphore mutex.

```
/* initialize mutex := 1 */
V(mutex);
   ...
critical section
   ...
P(mutex);
```

* Violate mutual exclusion

b) Suppose that process exchanges V(mutex) with P(mutex).

```
P(mutex);
   ...
critical section
   ...
P(mutex);
```

*** A deadlock will occur.**

c) Suppose that a process omits the P(mutex) or the V(mutex) or both

*** Violate mutual exclusion or a deadlock will occur**
Note: This time dependent errors give a motivation of language constructs which has an abstract data type.

2. CLASSICAL PROCESS COORDINATION PROBLEMS

1) THE BOUNDED-BUFFER PROBLEMS

```
TYPE item = ...;
VAR buffer = ...;
     full, empty, mutex: semaphore;
     nextp, nextc: item;

     /* mutex provide mutual exclusion for access to the buffer pool; empty and full
        count the number of empty and full buffers respectively */

BEGIN
  full := 0;
  empty := n;
  mutex := 1;

  PARBEGIN
    PRODUCER: REPEAT
                 ...
               produce an item in nextp;
                 ...
               P(empty);
               P(mutex);
                 ...
               add nextp to buffer
                 ...
               V(mutex).
               V(full);
          UNTIL false;
  CONSUMER: REPEAT
                 P(full);
                 P(mutex);
                    ...
                 remove an item from buffer to nextc
                    ...
                 V(mutex);
                 V(empty);
                    ...
                 consume the item in nextc
                    ...
              UNTIL false;
  PARAEND;
END;
```

3. INTERPROCESS COMMUNICATION

1) Shared-memory

Shared-memory system require communicating processes to share some variables. The process are expected to exchange information through the use of these shared variables In a shared memory scheme the responsibility for providing communication rests with the application programmers; the operating system only needs to provide the shared memory.

2) Message-system

a) What is message-system ?

Message scheme allows the processes to exchange messages. In this scheme, the responsibility for providing communication rests with the operating system itself. An interprocess communication facility provide two operations: send(message) and receive(message).

b) Logical implementation questions.

*** How can links established?**
*** How many links can there be between every pair of processes?**
*** What is the capacity of a link?**
*** Is a link unidirectional or bidirectional?**

c) Methods for logically implementing a link and the SEND/RECEIVE operations:

*** Direct or indirect communication.**
*** Send to a process or to a mailbox.**
*** Symmetric or asymmetric communication.**
*** Automatic or explicit buffering.**
*** Send by copy or send by reference.**
*** Fixed-size or variable-size messages.**