


CS-3103 : Operating Systems : Sec-A (NB)

Process Synchronization – Semaphores & Monitors

OPERATING
SYSTEM



Computer Operating Systems: OS Families for Computers

Semaphores

■ A semaphore is a non-negative integer that can be operated upon by two atomic operators, '**wait**' and '**signal**'. A semaphore (say s) can be initialized to a non-negative integer value.

- **signal (s):** This operator increments the semaphore value by 1. The increment operation is indivisible, i.e., no two processes can simultaneously increment the value of the semaphore. Example: let us assume that the current value of the semaphore s is 4. Now, if two processes P1 and P2 compete to perform the operation 'signal(s)', the value of the semaphore s will be 6. This means each process will signal the semaphore s . The first and second processes will increment the semaphore(s) to 5 and 6 respectively.
- **wait(s):** This operator decrements the semaphore value by 1 as long as the value of the semaphore is more than 0. This means that if the value of semaphore is 0 then the 'wait' operation will have no effect, i.e., the semaphore value will remain unchanged at 0. The decrement operation is also indivisible, i.e., no two processes can simultaneously decrement the value of the semaphore.

Semaphores

The behavior of the wait and signal operators are given below:

wait (s): if ($s > 0$), then decrement s

signal (s): increment s

The ‘**wait**’ and ‘**signal**’ operators are used to **guard** a CS with the help of a semaphore. The arrangement is given below:

wait (s)

{ CS }

Fig A

signal (s)

parbegin

$p1, p2, p3, \dots, pn$; \Rightarrow discussion in there in next slide

parend

Semaphores

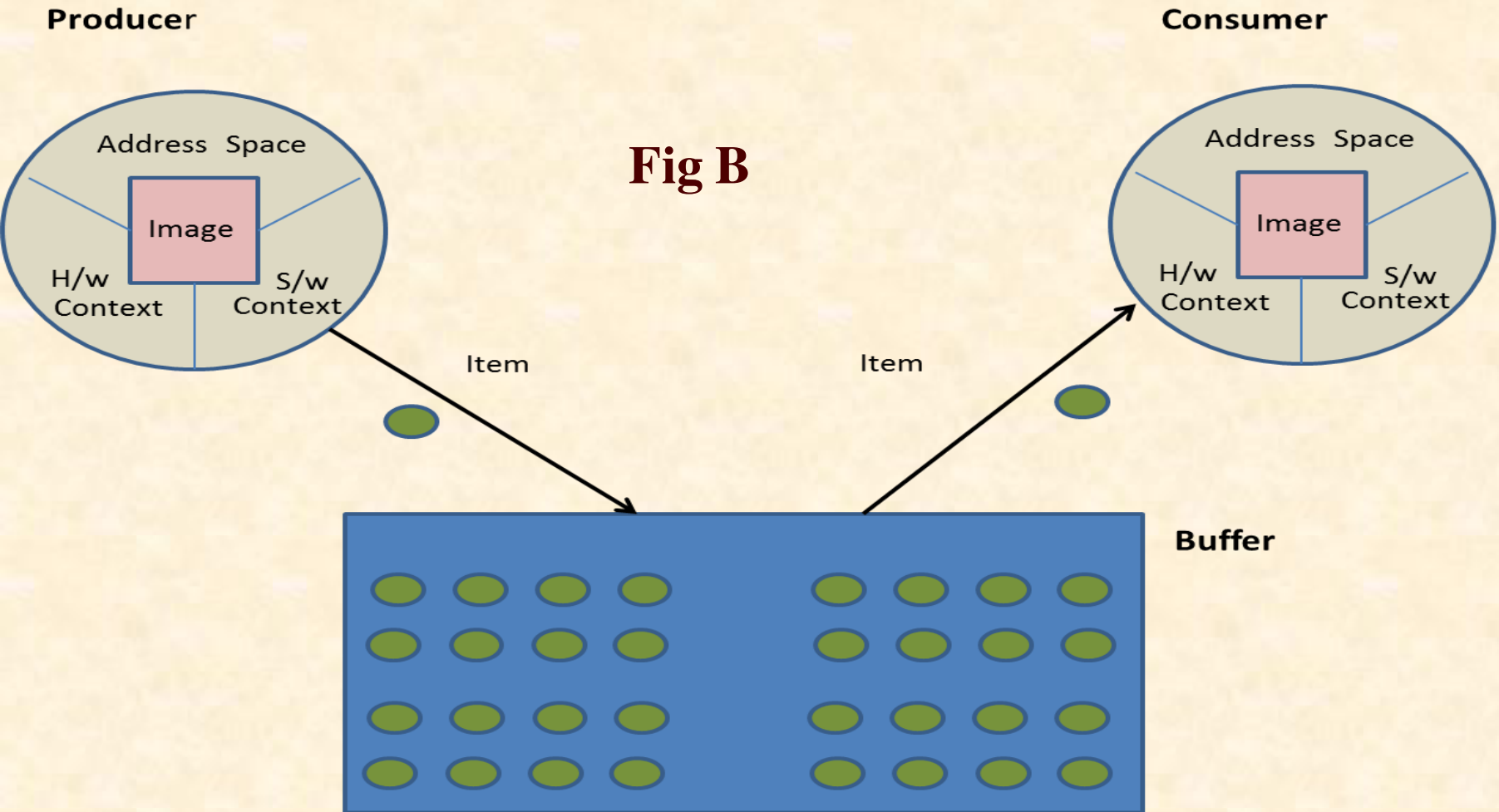
It may be noted from Fig. A, that processes $p_1, p_2, p_3, \dots, p_n$ are concurrent processes. All are trying to access the CS. A process can enter the CS after successfully executing the 'wait' operation. The number of processes that can enter the CS is limited by the value of semaphore ' s '. If $s = N$, then only N processes can enter the CS at a time. However, if $s = 1$, then only one process can enter the CS and the rest of the processes will find the semaphore value equal to 0, and must wait. As soon as a process exits from the CS, it executes the 'signal' operator resulting in an increment in the semaphore value, i.e., a 'signal' operation sends a signal through the semaphore that at least one process can enter the CS.

Categories of Semaphores

- **Binary semaphore:** This type of semaphore is initialized with values of only 1 or 0. It allows only one process to execute a 'wait' operation.
- **Counting semaphore:** This semaphore can take any non-negative values, i.e., any value greater than or equal to 0. It is generally used for synchronization purpose and especially to guard a resource.

Working of Semaphores by producer – consumer problem

■ A producer process produces some item while a consumer process consumes the item.



A buffer between a producer and consumer

Working of Semaphores by producer – consumer problem

- ❑ There are two types of processes in a system, the producer process that produces some item, and the consumer process that consumes the item.
- ❑ There may be a mismatch between the speeds of producers and consumers. Either the producer or the consumer is faster.
- ❑ For example, a CPU is faster than a printer. A currently executing process can generate data at a tremendous speed while the printer driver on the other hand cannot consume the data at the same rate, because the printer is a comparatively slow device.
- ❑ In order to manage the speed mismatch, the system introduces a buffer between the producer and the consumer. The producer produces the items and stores them in the buffer. The consumer extracts the items from buffer and consumes them (**Fig B**).

Working of Semaphores by producer – consumer problem

- ❑ However, the producer and the consumer cannot enter the buffer simultaneously, i.e., they mutually exclude each other.
- ❑ We can restrict entry into the buffer by introducing a semaphore called 'buffAvail' that is initialized as '1', indicating that only one process can enter the buffer at a time.
- ❑ The producer and consumer are coded as do-forever processes.
- ❑ *Producer → The job of the producer is to produce an item and wait for the buffer to be available.*
- ❑ *If the buffer is available, it enters the buffer and stores the item into it and comes out.*
- ❑ *While coming out of the buffer, the producer sends a signal to the consumer that it is out of the buffer by executing the 'signal' operator.*
- ❑ *Consumer → The consumer waits for the buffer to be available. If the buffer is available, the consumer enters the buffer, extracts the item from it, and comes out.*
- ❑ *While coming out of the buffer, the consumer sends a signal to the producer that it is out of buffer, by executing the 'signal' operator. (Fig C)*

Working of Semaphores by producer – consumer problem

buffAvail = 1;

Fig C

producerProc

ConsumerProc

```
while (1)
{
```

```
    produce an item;
```

```
    wait (buffAvail);
```

```
    enter buffer store the item;
```

```
    signal (buffAvail);
```

```
    perform rest of the work;
```

```
}
```

```
while (1)
{
```

```
    wait (buffAvail);
```

```
    enter buffer extract the item;
```

```
    signal (buffAvail);
```

```
    consume the item
```

```
    perform rest of the work;
```

```
}
```

.....
Synchronization signal



START

Monitors

- A monitor is a programming construct used for process synchronization.
- It has four components:
 - Local data
 - Monitor procedures
 - Initialization code
 - Monitor entry queue
- ✓ The data inside the monitor is of two types, global and local. The global data is available to all monitor procedures and the local data is local to a specific monitor procedure.
- ✓ *Note*: No data item is accessible from outside the monitor, thus ensuring data and information hiding.

Entry Queue of
processes



Data

Monitor Proc 1

Monitor Proc 2

Monitor Proc n

Initialization code

**Structure of a
monitor**

Monitors

- ❑ A *monitor procedure* can be called by a process from outside the monitor. ***The procedure guards a resource by a pair of wait and signal operations.***
 - ❑ A process enters the monitor by calling a monitor procedure.
 - ❑ Only one process is allowed to enter the monitor while other processes are made to wait in the entry queue of the monitor.
 - ❑ ***When a process leaves the monitor, it sends a signal whereby a waiting process is allowed to enter.***
 - ❑ If no process is waiting then the signal has no effect.
 - ❑ The wait and signal operations work on conditional variables.
-
- ❑ Let us now consider a “***process synchronization***” problem called a ***reader-writer*** problem. In this problem, the reader and writer processes attempt to access a file.
 - ❑ *Readers and writers mutually exclude each other, i.e., simultaneous read and write operations cannot take place on a file.*
 - ❑ However, *more than one reader can read a file at a time.*
 - ❑ As far as writers are concerned, *only one writer can write into the file.* Let the name of the monitor be ‘***fileAccess***’.

Monitors

Monitor fileAccess

```
{
    int          numReader;
    boolean readMode;
    condition writer;
    void openReader ()
    {
        while ( !readMode ) { do nothing }
        numReader = numReader + 1;
        if (numReader == 1)
            { wait (writer);
              readMode = true;}
    }
    void closeReader()
    {
        numReader = numReader - 1;
        if (numReader == 0)
            { signal (writer);
              }
    }
    void openWriter ()
    {
        while (readMode) && (numReader > 0) [do nothing]
        wait (writer);
        readMode = false;
    }
    void closeWriter ()
    {
        signal (writer);
        readMode = true;
    }
}

{
    numReader=0;
    raedMode = true;
}
}
```

The diagram illustrates the semaphore operations within the Monitor fileAccess code. A yellow box labeled "semaphore" is connected to the "wait (writer)" and "signal (writer)" calls in the code. Arrows indicate the flow of control between the semaphore and the monitor's internal state.

Operations done by '*fileAccess*' Monitor:

1. **numReader**: It is an integer variable that keeps the count of the number of readers which are currently reading the file. It is initially set to **0**.
2. **readMode**: It is a Boolean variable. It is initially set to true, indicating that the file is available for reading. When a writer enters the file for writing, it sets readMode to **false**. As and when a writer comes out of the file, it sets readMode to true.
3. **writer**: It is a conditional variable on which a process applies 'wait' or 'signal' for entering into or leaving the file.
4. **openReader()**: It is a procedure that a reader process must call before entering the file for reading. If it is the first reader, then it must ensure that the writer is out of the file before it starts reading the file.
5. **closeReader()**: It is a procedure that a reader must call before leaving the file. If it is the last reader, then it must send the signal to the writer that the file is available for writing.
6. **openWriter()**: It is a procedure that a writer process must call before entering the file for writing.
7. **closeWriter()**: It is a procedure that a writer process must call before leaving the file after writing..