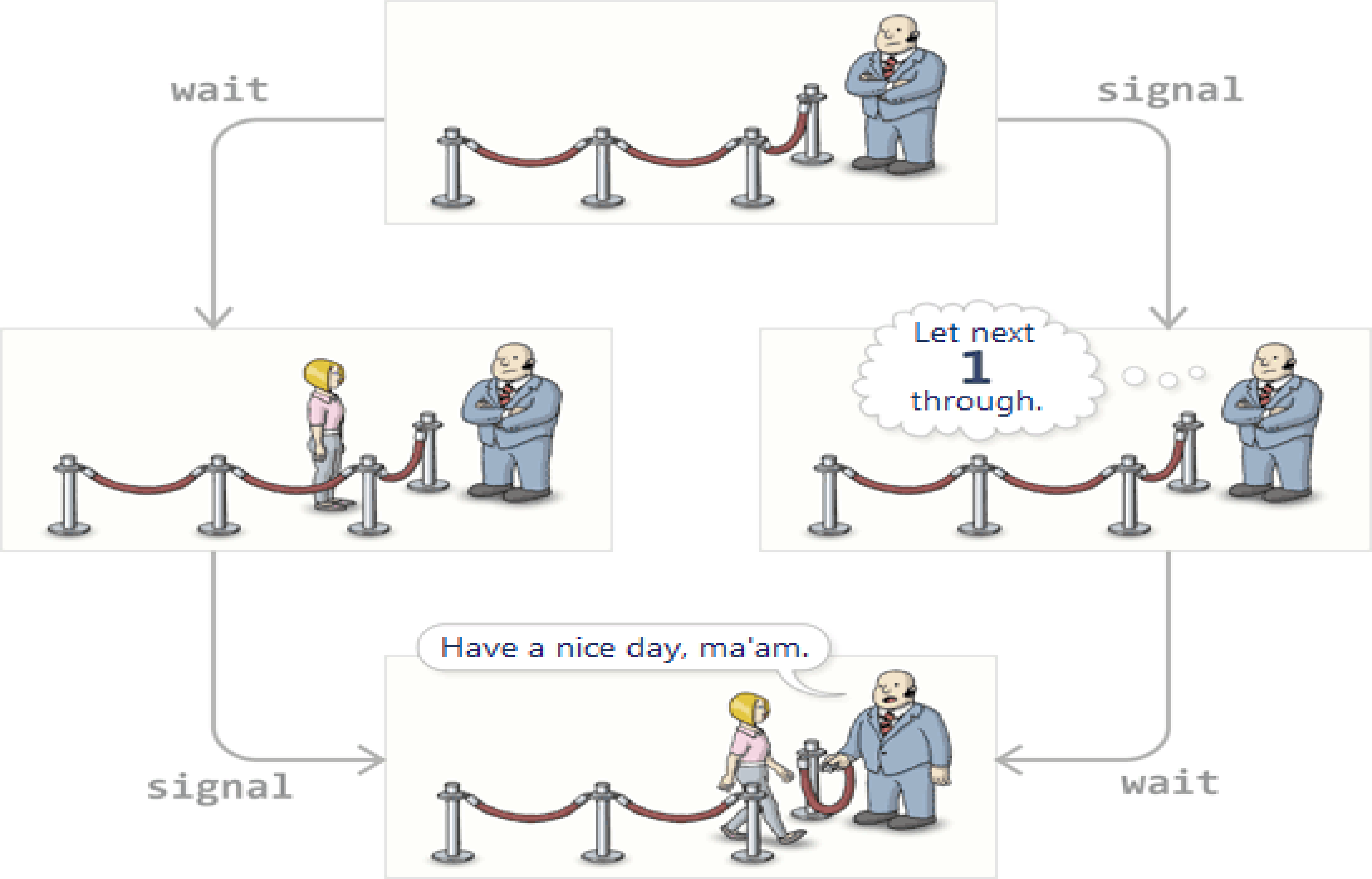
CS-3103 : Operating Systems : Sec-A (NB)

Process Synchronization – Semaphores & Monitors



Semaphores

- A semaphore is a non-negative integer that can be operated upon by two atomic operators, ‘**wait**’ (P) and ‘**signal**’ (V). 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) // drecements → --

{ CS}

Fig A

signal (s) // increments → ++

parbegin

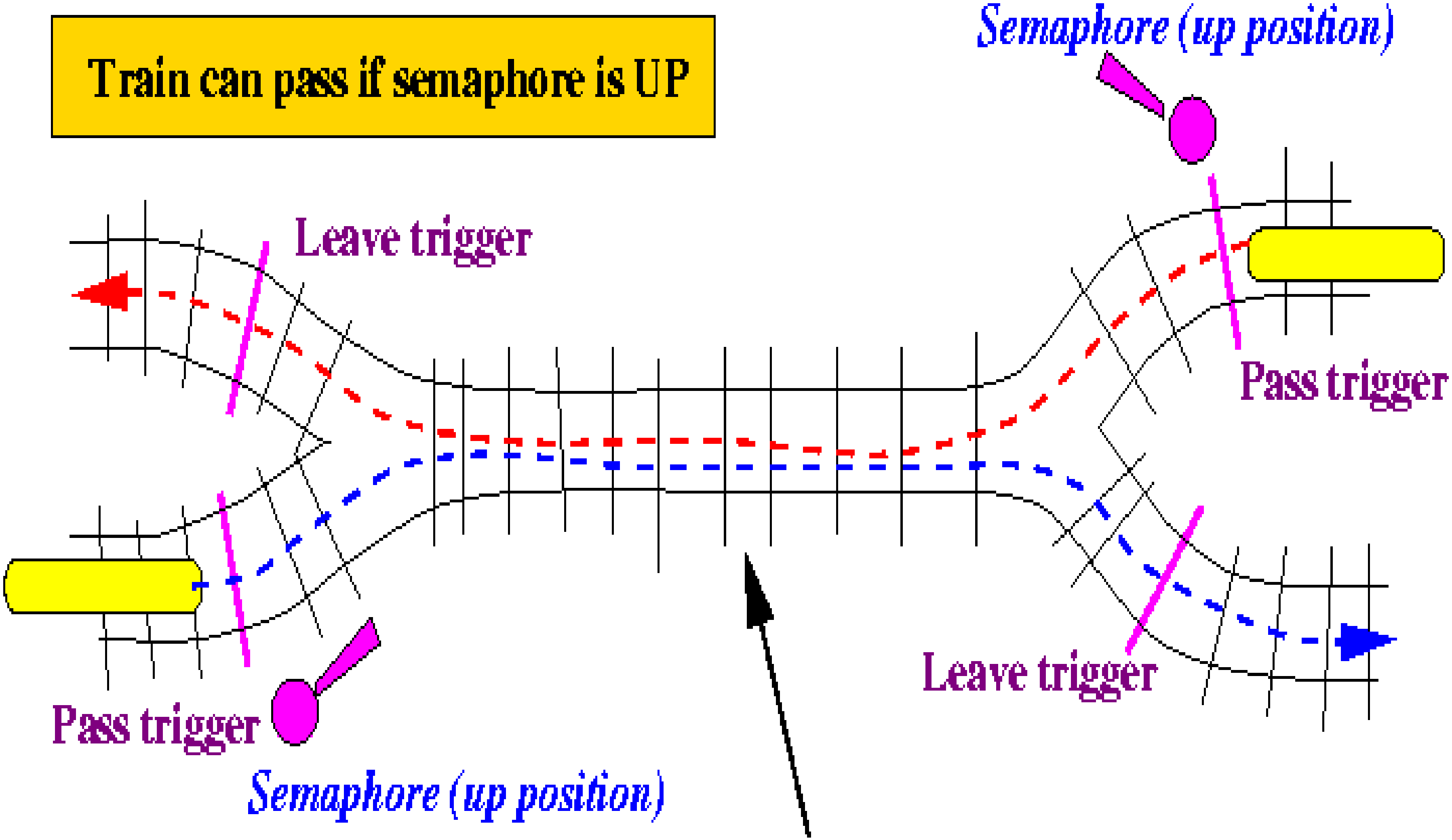
p1, p2,p3,…..pn;* → *discussion in there in next slide ….

parend

Semaphores

It may be noted from Fig. A, that processes p1, p2, p3….pn 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.

Train can pass if semaphore is UP



Never allow more than one train on this part of the track !!!

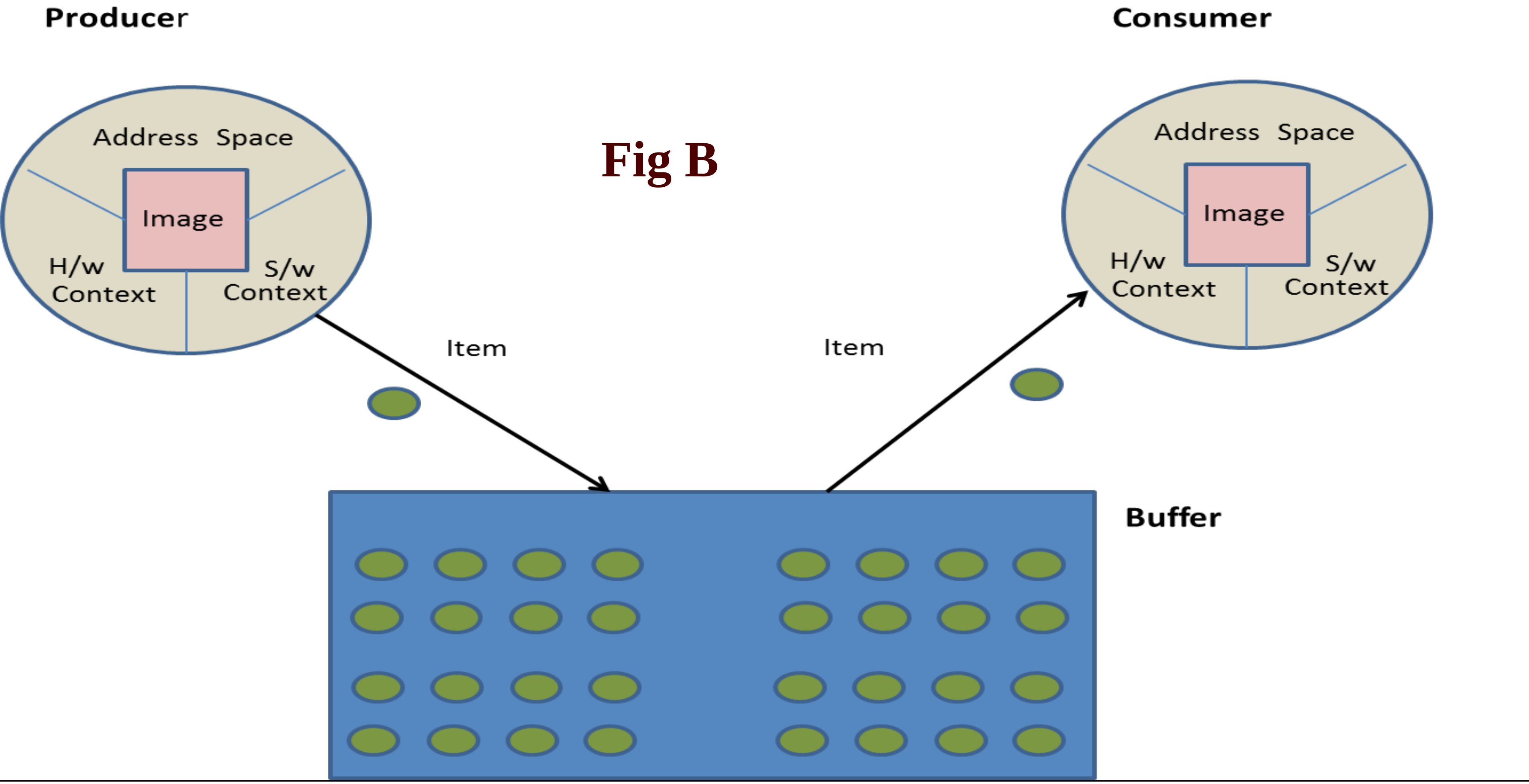
Categories of Semaphores

- **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.

START

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 (P) and the consumer (C).*** The producer produces the items and stores them in the buffer. The consumer extracts the items from buffer and consumes them (**Fig B).**

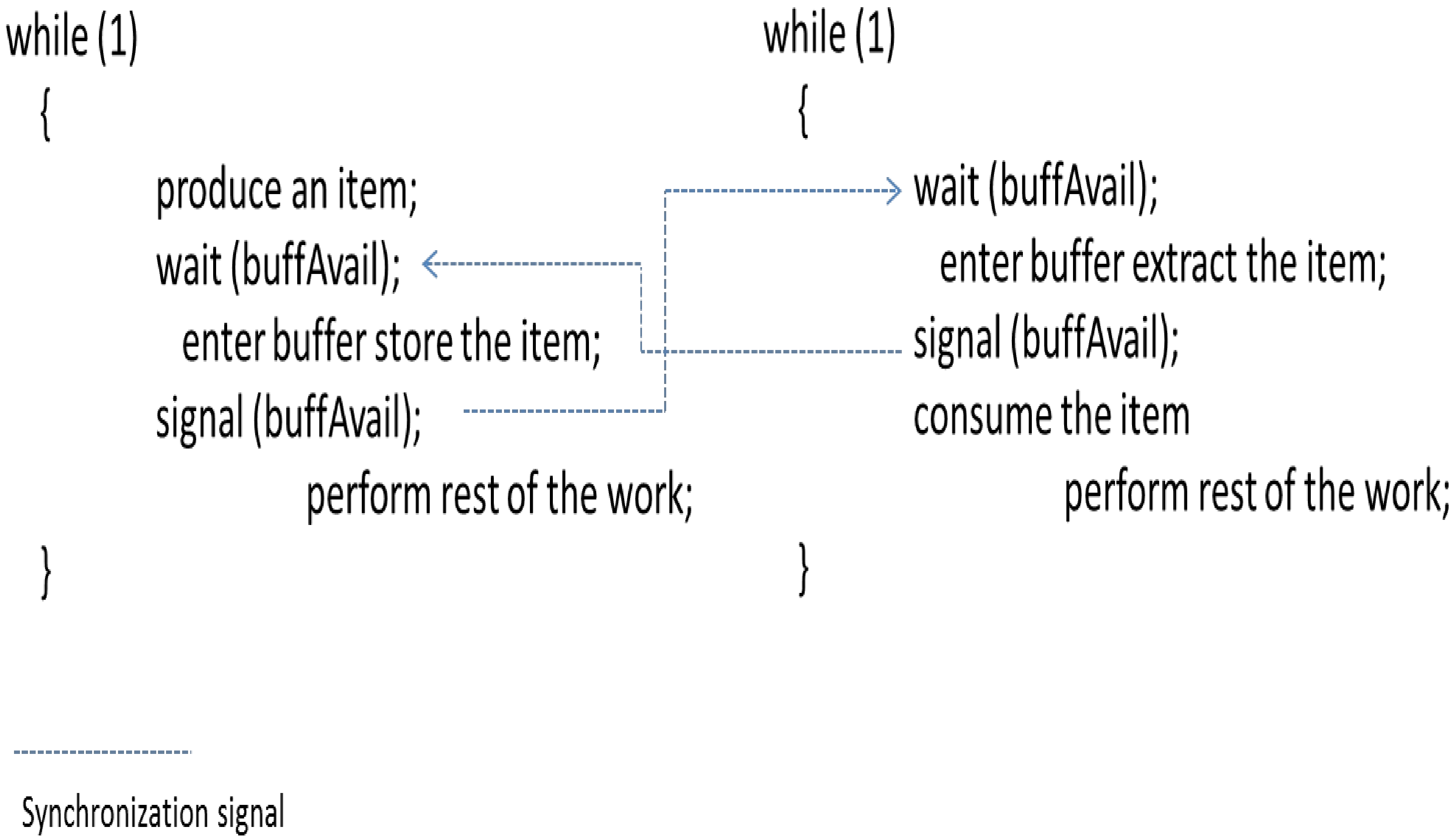
Only ONE Buffer between P & C

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

Fig C



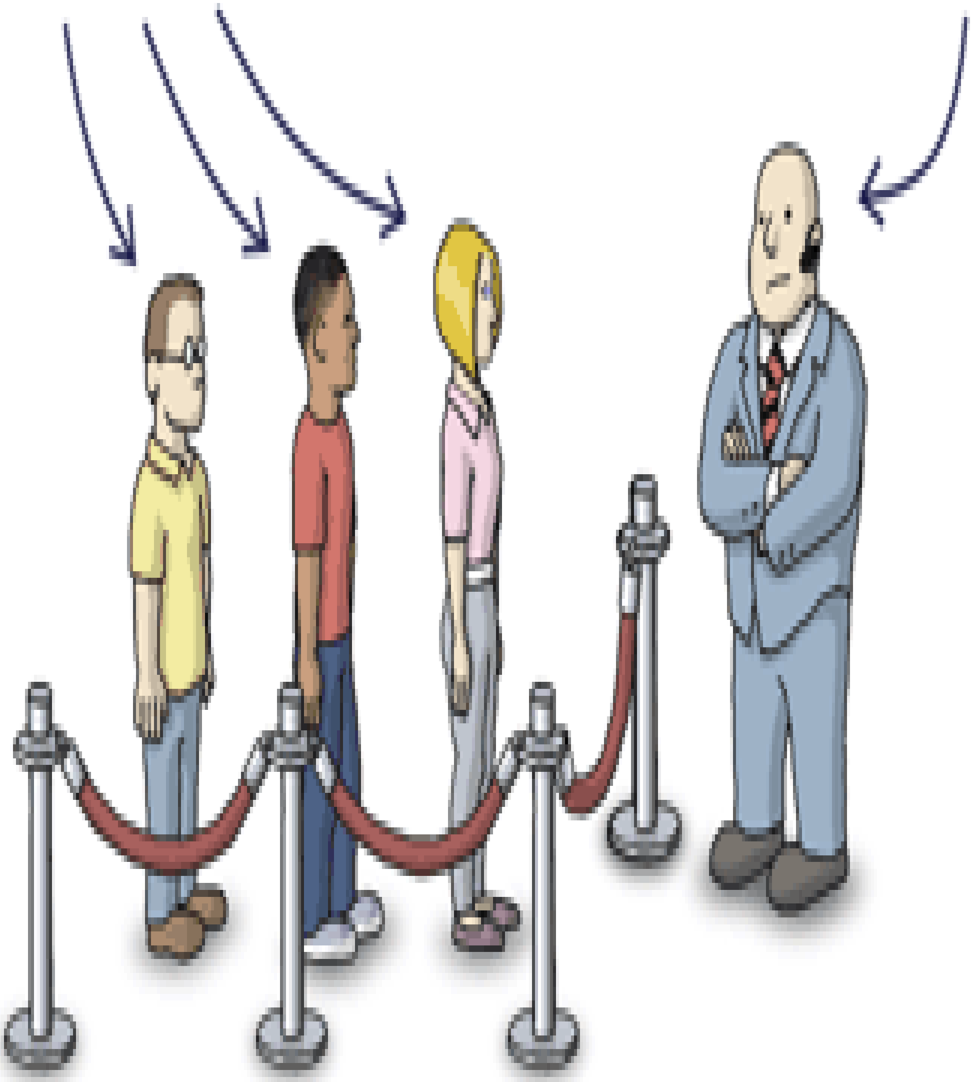
```
                    buffAvail = 1;

producerProc

while (1)
  {
        produce an item;
        wait (buffAvail);
          enter buffer store the item;
        signal (buffAvail);
                perform rest of the work;
  }

ConsumerProc

while (1)
  {
        wait (buffAvail);
          enter buffer extract the item;
        signal (buffAvail);
        consume the item
                perform rest of the work;
  }
```



These people represent **waiting threads.** They aren't running on any CPU core.

The bouncer represents a **semaphore.** He won't allow threads to proceed until instructed to do so.

Solving synchronization problems using Semaphores

- A semaphore :
  a) is a binary mutex
  b) must be accessed from only one process
  c) can be accessed from multiple processes
  d) None of these

- Answer: c

Solving synchronization problems using Semaphores

- The two kinds of semaphores are : (choose two)
  a) mutex
  b) binary
  c) counting
  d) decimal

- Answer: b and c

Solving synchronization problems using Semaphores

- At a particular time of computation the value of a counting semaphore is 7.Then 20 P operations and 15 V operations were completed on this semaphore.The resulting value of the semaphore is :
  a) 42
  b) 2
  c) 7
  d) 12

- Answer: b
  Explanation: P represents Wait and V represents Signal. P operation will decrease the value by 1 everytime and V operation will increase the value by 1 everytime.

Solving synchronization problems using Semaphores

- A binary semaphore is a semaphore with integer values : (choose two)
  a) 1
  b) -1
  c) 0
  d) 0.5

- Answer: a and c
  Explanation: None

❑ **Semaphores are very useful in process synchronization and multithreading.**

❑ The POSIX system in Linux presents its own built-in semaphore library (**semaphore.h**).

1. To lock a semaphore or wait we can use the sem_wait function:

int sem_wait(sem_t *sem);

2. To release or signal a semaphore, we use the sem_post function:

int sem_post(sem_t *sem);

3. A semaphore is initialised by using sem_init(for processes or threads) or sem_open (for IPC): sem_init(sem_t *sem, int pshared, unsigned int value);

Where,

sem : Specifies the semaphore to be initialized.

pshared : This argument specifies whether or not the newly initialized semaphore is shared between processes or between threads. A non-zero value means the semaphore is shared between processes and a value of zero means it is shared between threads.

value : Specifies the value to assign to the newly initialized semaphore.

```
sem_t mutex;
sem_init(&mutex,0,1);
sem_getvalue(&mutex,&n);
printf("binary semaphore value=%d\n",n);
```

4. To destroy a semaphore, we can use sem_destroy.
sem_destoy(sem_t *mutex);

5. To declare a semaphore, the data type is sem_t.

Solve the producer consumer problem for a single piece of data using binary

```
#include<stdio.h>              #include<stdlib.h>
#include<semaphore.h>     #include<pthread.h>     #include<signal.h>

sem_t   mutex;
int pizza=0, counter=0, n;

void func(int s) {
    printf("Signal received SIGQUIT = %d trying to terminate by Ctrl+\\ \n",s);
    signal(SIGQUIT,SIG_DFL);
 }

void func1(int s) {
    printf("Signal received SIGTSTP = %d trying to terminate by Ctrl+Z \n",s);
    signal(SIGTSTP,SIG_DFL);
  }

void func2(int s) {
    printf("Signal received SIGINT = %d trying to terminate by Ctrl+C \n",s);
    signal(SIGINT,SIG_DFL);
  }
```

```
int main()
{
        signal(SIGQUIT,func); signal(SIGTSTP,func1); signal(SIGINT,func2);
        pthread_t th_p,th_c;
        int retval;
        sem_init(&mutex,0,1);
        sem_getvalue(&mutex,&n);
        printf("binary semaphore value=%d\n",n);
        retval=pthread_create(&th_p, NULL, producer, NULL);
         if(retval!=0)  exit(-1);
         printf("\nAfter creating producer thread");
        retval=pthread_create(&th_c, NULL, consumer, NULL);
         if(retval!=0)  exit(-1);
        retval=pthread_join(th_p,NULL);
         if(retval!=0)  exit(-1);
         retval=pthread_join(th_c,NULL);
         if(retval!=0)  exit(-1);
         sem_destroy(&mutex);
         return 0;
}
```

```
void *producer(void *s)  {
 while(1)
   {   sleep(4);
       sem_wait(&mutex);
       sem_getvalue(&mutex,&n);
       pizza++; counter++;
      printf("\n Producer: With counter (%d) pizza made=%d with semaphore
wait value = %d \n",counter,pizza,n);
      sem_post(&mutex);
      sem_getvalue(&mutex,&n);
     printf("\n Producer: Semaphore post (signal) value = %d \n",n);
  }
   return ;
}
```

```
void * consumer(void *s)
  {
     while(1)
    {
      sleep(5);
      sem_wait(&mutex);
      sem_getvalue(&mutex,&n);
      pizza--; counter++;
      printf("\n Consumer: With counter (%d) pizza consumed and Amount left=
%d with semaphore wait value = %d \n",counter,pizza,n);
      sem_post(&mutex);
      sem_getvalue(&mutex,&n);
      printf("\n Consumer: Semaphore post (signal) value = %d \n",n);
   }
   return NULL;
}
```

START

Mutex vs Semaphore

Mutex:

Chef is making Burger one at a time. When finished, the chef place the pizza in the *cabinet* and gives signal to the buyer in the queue to buy the item and collect the pizza from cabinet. Just One-to-One relationship.

Officially: “Mutexes are typically used to serialize access to a section of re-entrant code that cannot be executed concurrently by more than one thread.

A mutex object only allows one thread into a controlled section, forcing other threads which attempt to gain access to that section to wait until the first thread has exited from that section.”

(A mutex is really a semaphore with value 1.)

Mutex vs Semaphore

Semaphore:

Is the number of identical *carts* to place foodies. Example, say we have four *cabinets* with identical locks and keys.

The semaphore count – the count of keys – is set to 4 at beginning (all four cabinets are full), then the count value is decremented as buyers are coming in to buy foodies. If all cabinets are full, then the semaphore count is 0. Now, when the chef started making burger again and starts filling up the cabinets, semaphore is increased to 1 and given to the next person (5th) in the queue.

Officially: “A semaphore restricts the number of simultaneous users of a shared resource up to a maximum number. Threads can request access to the resource (decrementing the semaphore), and can signal that they have finished using the resource (incrementing the semaphore).”

Question

- The following pair of processes share a common variable X :

  ```
  Process A
  int Y;
  A1: Y = X*2;
  A2: X = Y;
  ```

- Process B

  ```
  int Z;
  B1: Z = X+1;
  B2: X = Z;
  ```

- X is set to 5 before either process begins execution. As usual, statements within a process are executed sequentially, but statements in process A may execute in any order with respect to statements in process B.

i) How many different values of X are possible after both processes finish executing ?

a) two

b) three

c) four

d) eight

Question continues…..

- i) Answer: c
  Explanation: Here are the possible ways in which statements from A and B can be interleaved.
  A1 A2 B1 B2: X = 11
  A1 B1 A2 B2: X = 6
  A1 B1 B2 A2: X = 10
  B1 A1 B2 A2: X = 10
  B1 A1 A2 B2: X = 6
  B1 B2 A1 A2: X = 12

Synchronization problems

- ii) Suppose the programs are modified as follows to use a shared binary semaphore T :

  ```
  Process A
  int Y;
  A1: Y = X*2;
  A2: X = Y;
  signal(T);
  ```

-
  ```
  Process B
  int Z;
  B1: wait(T);
  B2: Z = X+1;
  X = Z;
  ```

T is set to 0 before either process begins execution and, as before, X is set to 5.

Now, how many different values of X are possible after both processes finish executing ?

a) one

b) two

c) three

d) four

Synchronization problems…

- Answer: a
  Explanation: The semaphore T ensures that all the statements from A finish execution before B begins. So now there is only one way in which statements from A and B can be interleaved:
  A1 A2 B1 B2: X = 11.

Solving synchronization problems using Semaphores

- Semaphores are mostly used to implement :
  a) System calls
  b) IPC mechanisms
  c) System protection
  d) None of these
  View Answer
- Answer: b

❑The bounded buffer problem is also known as :

a) Readers – Writers problem

b) Dining – Philosophers problem

c) Producer – Consumer problem

d) None of these

Answer: c

❑In the bounded buffer problem, there are the empty and full semaphores that :

a) count the number of empty and full buffers

b) count the number of empty and full memory spaces

c) count the number of empty and full queues

d) None of these

Answer: a

In the bounded buffer problem :
a) there is only one buffer
b) there are n buffers (n being greater than one but finite)
c) there are infinite buffers
d) the buffer size is bounded **Answer: b**

To ensure difficulties do not arise in the readers – writers problem, ________ are given exclusive access to the shared object.
a) readers
b) writers **Answer: b**
c) None of these

The dining – philosophers problem will occur in case of :

a) 5 philosophers and 5 chopsticks

b) 4 philosophers and 5 chopsticks

c) 3 philosophers and 5 chopsticks

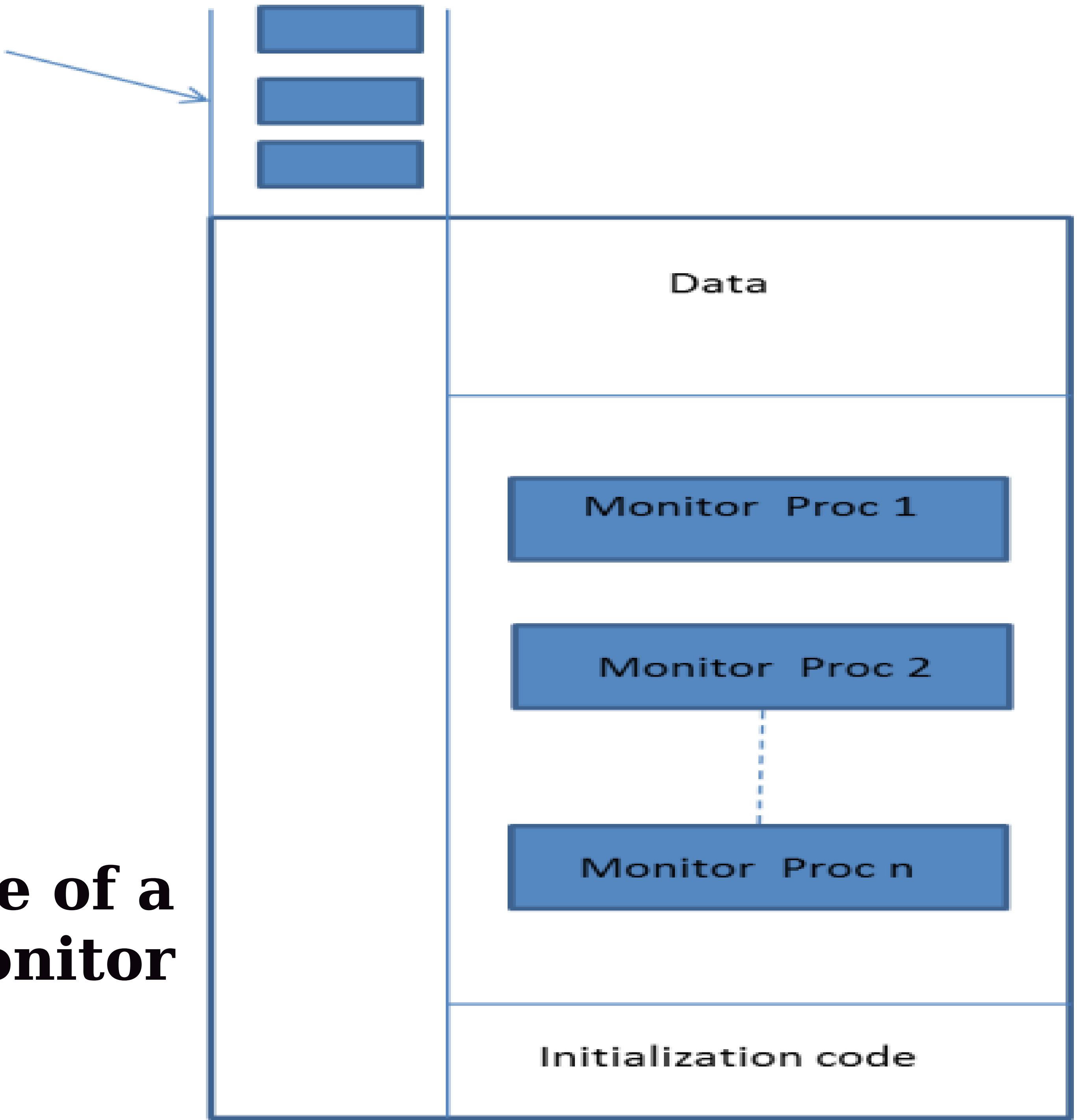
d) 6 philosophers and 5 chopsticks

Answer: a

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.

Structure of a monitor

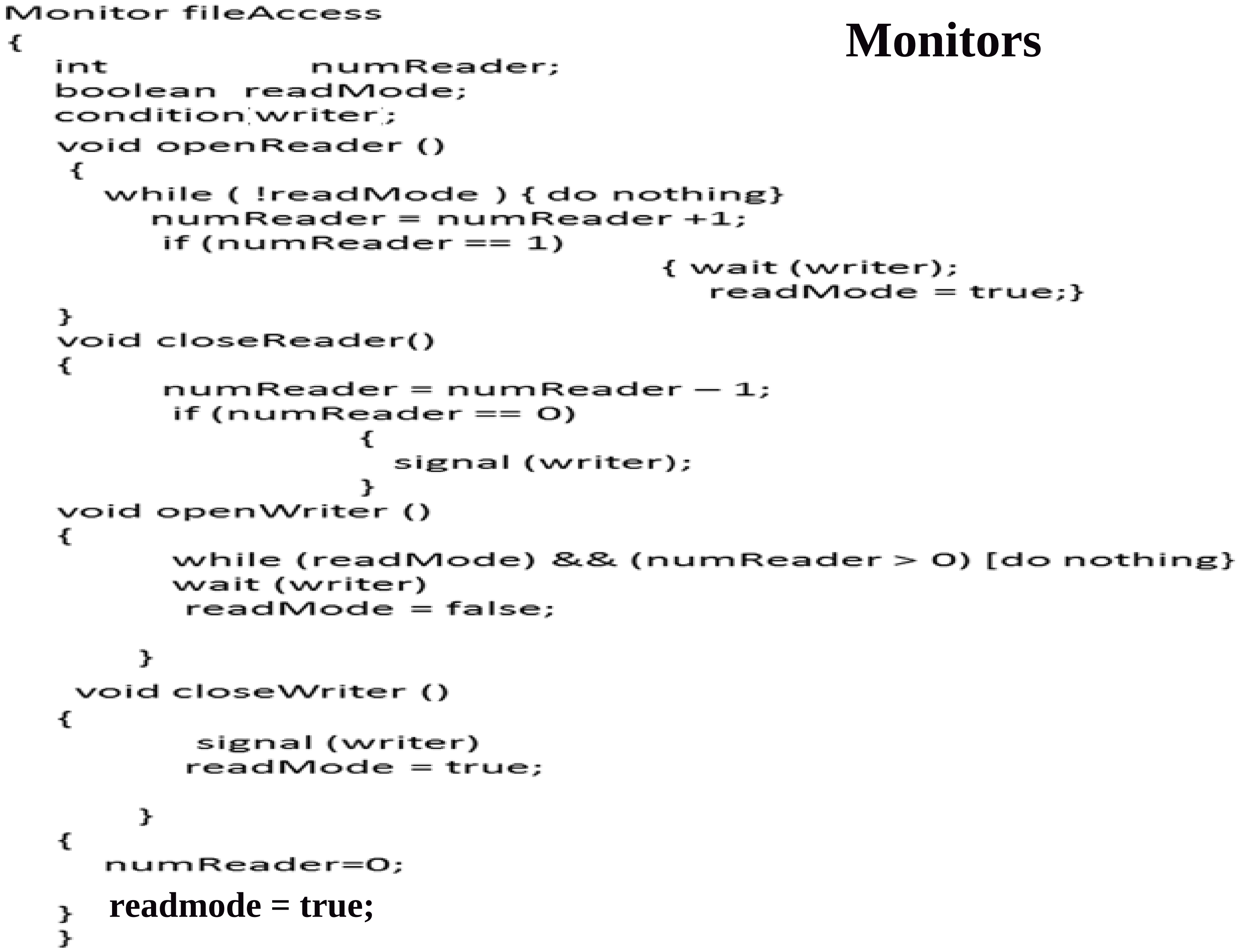


Monitors

- 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;
    }  readmode = true;
    }
```

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..

STOP AND TAKE PAUSE