BCA Fourth Semester

Operating Systems

Unit -3 Process Management (Part 2) [15 Hrs]

Problems with mutual Exclusion:

- **The above techniques achieves the mutual exclusion using busy waiting**.
- Here while one process is busy updating shared memory in its critical region, no other process will enter its critical region and cause trouble.

- **Mutual Exclusion with busy waiting just check to see if the entry is allowed when a process wants to enter its critical region, if the entry is not allowed the process just sits in a tight loop waiting until it is**

1. This approach waste CPU time
2. There can be an unexpected problem called priority inversion problem.

Priority Inversion Problem:

- **Consider a computer with two processes, H, with high priority and L, with low priority, which share a critical region.**
- The scheduling rules are such that H is run whenever it is in ready state.
- At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes).
- H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever.
- This situation is sometimes referred to as **the priority inversion problem**.

Let us now look at some IPC primitives that blocks instead of wasting CPU time when they are not allowed to enter their critical regions.

Using blocking constructs greatly improves the CPU utilization.

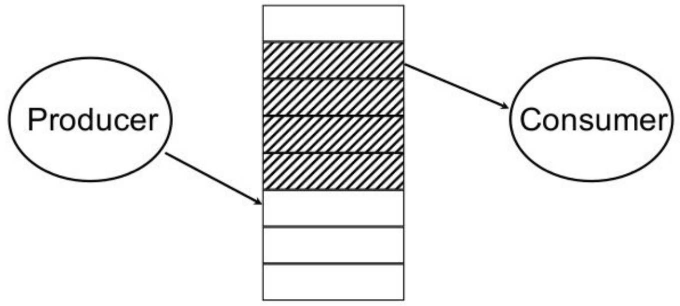
Sleep and Wakeup:

- **Sleep and wakeup are system calls that blocks process instead of wasting CPU time when they are not allowed to enter their critical region.**
- Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up.
- The wakeup call has one parameter, the process to be awakened.

Examples to use Sleep and Wakeup primitives:

Producer-consumer problem (Bounded Buffer):

- **Two processes share a common, fixed-size buffer. One of them, the producer, puts information into the buffer, and the other one, the consumer, takes it out.**

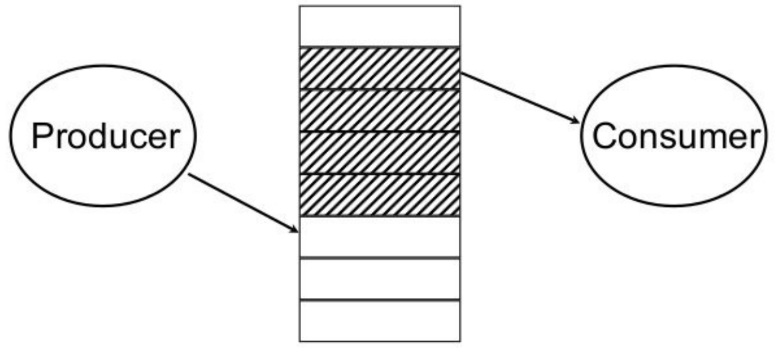


Bounded Buffer (cont.):

Trouble arises when

1. The producer wants to put a new data in the buffer, but buffer is already full. **Solution**: Producer goes to sleep and to be awakened when the consumer has removed data.

2. The consumer wants to remove data the buffer but buffer is already empty. **Solution**: Consumer goes to sleep until the producer puts some data in buffer and wakes consumer up.



Bounded Buffer (cont.):

N → Size of Buffer

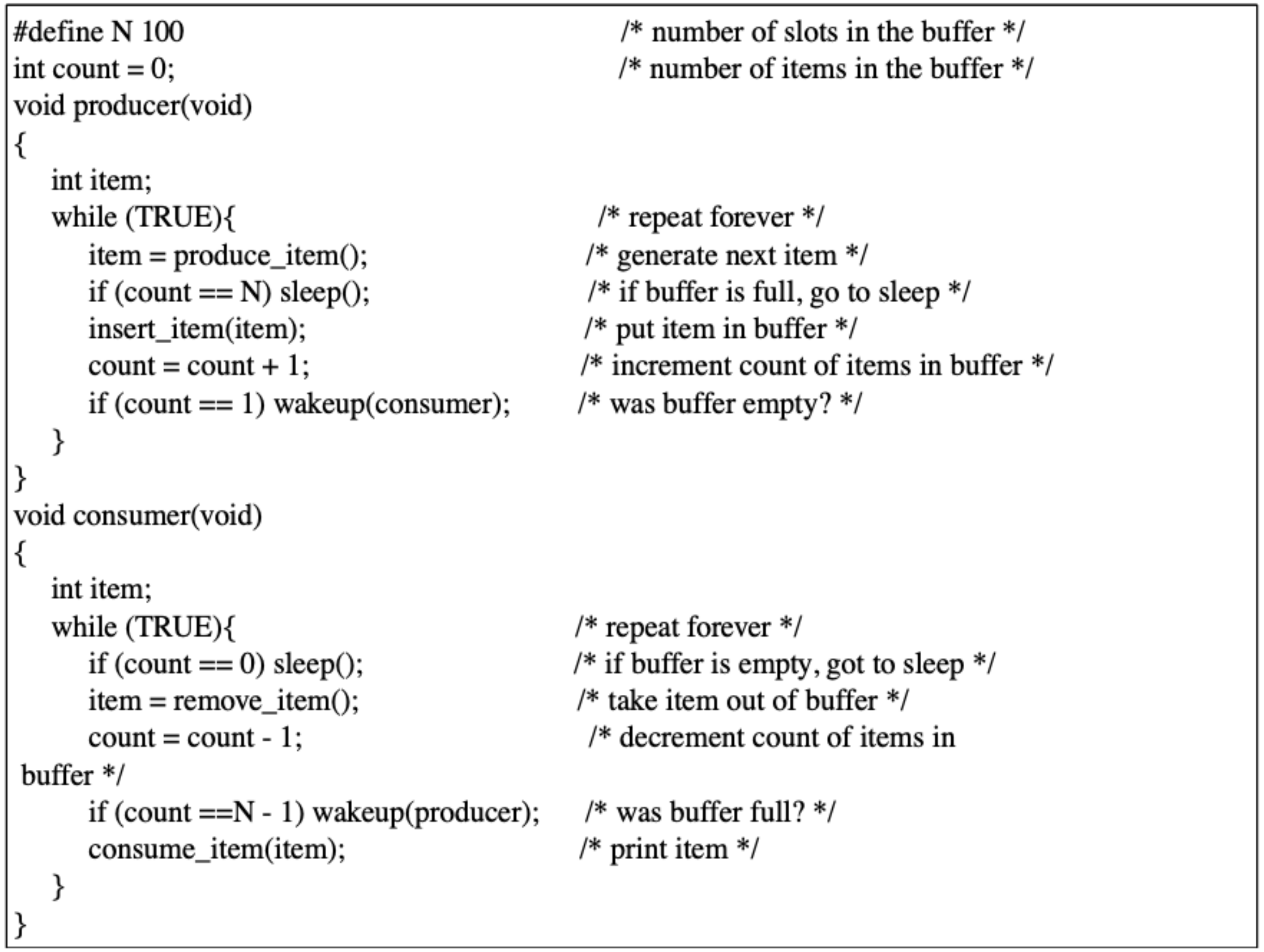
Count--> a variable to keep track of the no. of items in the buffer.

Producers code:

- The producers code is first test to see if count is N.
- If it is, the producer will go to sleep ; if it is not the producer will add an item and increment count.

Consumer code:

- It is similar as of producer. First test count to see if it is 0. If it is, go to sleep; if it nonzero remove an item and decrement the counter.
- Each of the process also tests to see if the other should be awakened and if so wakes it up.
- **This approach sounds simple enough, but it leads to the race conditions as we saw in the previous examples.**



```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */
void producer(void)
{
    int item;
    while (TRUE){                               /* repeat forever */
        item = produce_item();                  /* generate next item */
        if (count == N) sleep();                /* if buffer is full, go to sleep */
        insert_item(item);                      /* put item in buffer */
        count = count + 1;                      /* increment count of items in buffer */
        if (count == 1) wakeup(consumer);       /* was buffer empty? */
    }
}
void consumer(void)
{
    int item;
    while (TRUE){                               /* repeat forever */
        if (count == 0) sleep();                /* if buffer is empty, got to sleep */
        item = remove_item();                   /* take item out of buffer */
        count = count - 1;                      /* decrement count of items in
 buffer */
        if (count ==N - 1) wakeup(producer);    /* was buffer full? */
        consume_item(item);                     /* print item */
    }
}
```

Fig:The producer-consumer problem with a fatal race condition.

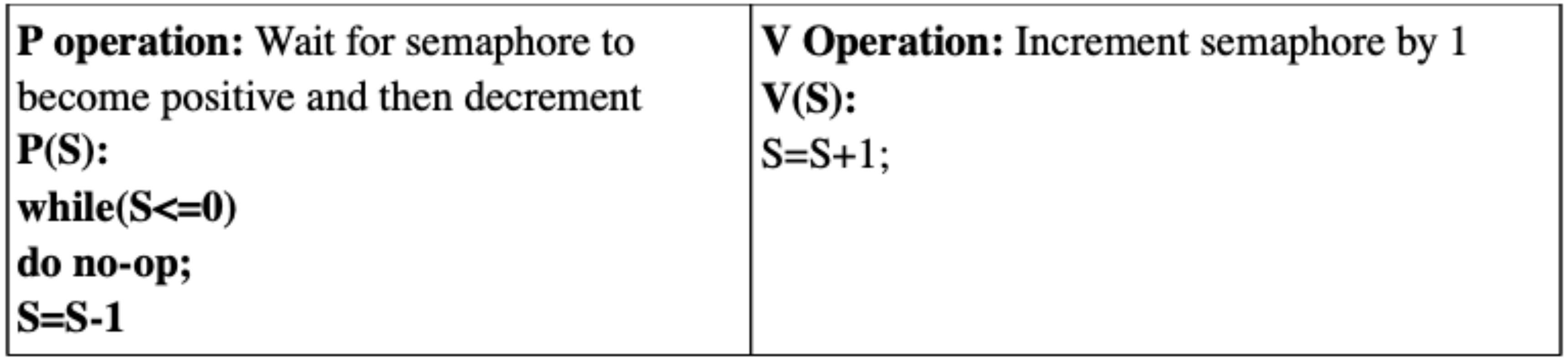
Bounded Buffer (cont.):

1. The buffer is empty and the consumer has just read count to see if it is 0.
2. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. (Consumer is interrupted and producer resumed)
3. The producer creates an item, puts it into the buffer, and increases count.
4. Because the buffer was empty prior to the last addition (count was just 0), the producer tries to wake up the consumer.
5. Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost.
6. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep.
7. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

The essence of the problem here is that a wakeup sent to a process that is not (yet) sleeping is lost. For temporary solution we can use wakeup waiting bit to prevent wakeup signal from getting lost, but it can't work for more processes.

Semaphore:

- **In computer science, a semaphore is a protected variable or abstract data type that constitutes a classic method of controlling access by several processes to a common resource in a parallel programming environment .**
- **A semaphore is a special kind of integer variable which can be initialized and can be accessed only through two atomic operations. P and V.**
- **If S is the semaphore variable, then,**



P operation: Wait for semaphore to become positive and then decrement	**V Operation:** Increment semaphore by 1
P(S):	**V(S):**
while(S<=0)	S=S+1;
do no-op;	
S=S-1	

Semaphore (cont.):

Atomic operations:

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

Semaphore operations:

P or Down, or Wait: P stands for *proberen* for "to test
V or Up or Signal: Dutch words. **V** stands for *verhogen* ("increase")

wait(sem) -- decrement the semaphore value. if negative, suspend the process and place in queue. (Also referred to as *P(), down* in literature.)

signal(sem) -- increment the semaphore value, allow the first process in the queue to continue. (Also referred to as *V(), up* in literature.)

Semaphore (cont.):

Counting semaphore – integer value can range over an unrestricted domain
Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
Also known as mutex locks .

Provides mutual exclusion

```
Semaphore S;    //  initialized to 1
wait (S);
        Critical Section
signal (S);
```

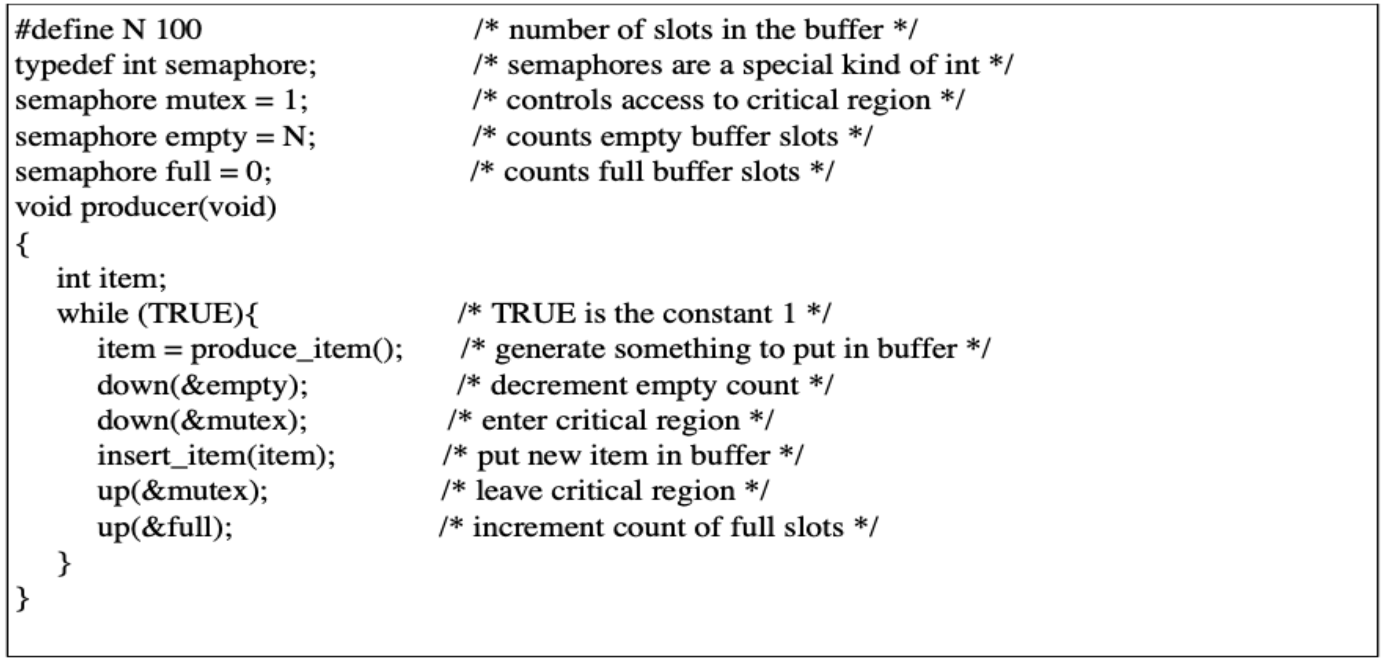
Semaphore implementation without busy waiting:
Implementation of wait:

```
        wait (S){
                value - -;
                if (value < 0) {
                        add this process to waiting queue
                         block();  }
                        }
```

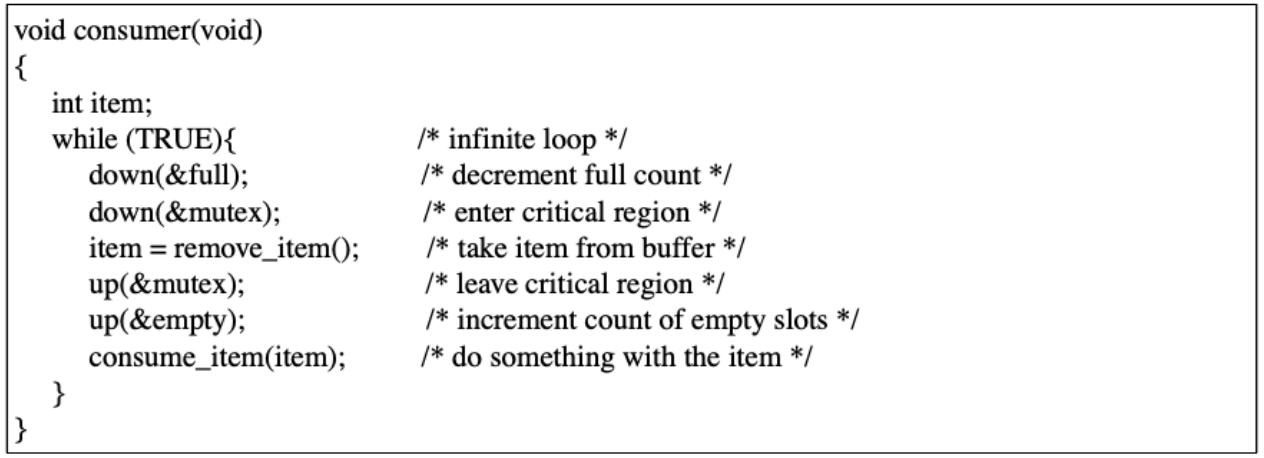
Implementation of signal:

```
        Signal (S){
                 value++;
                 if (value <= 0) {
                        remove a process P from the waiting queue
                         wakeup(P);  }
                    }
```

Semaphore (cont.):



```
#define N 100                      /* number of slots in the buffer */
typedef int semaphore;             /* semaphores are a special kind of int */
semaphore mutex = 1;               /* controls access to critical region */
semaphore empty = N;               /* counts empty buffer slots */
semaphore full = 0;                /* counts full buffer slots */
void producer(void)
{
    int item;
    while (TRUE){                  /* TRUE is the constant 1 */
        item = produce_item();     /* generate something to put in buffer */
        down(&empty);              /* decrement empty count */
        down(&mutex);              /* enter critical region */
        insert_item(item);         /* put new item in buffer */
        up(&mutex);                /* leave critical region */
        up(&full);                 /* increment count of full slots */
    }
}
```



```
void consumer(void)
{
    int item;
    while (TRUE){                  /* infinite loop */
        down(&full);               /* decrement full count */
        down(&mutex);              /* enter critical region */
        item = remove_item();      /* take item from buffer */
        up(&mutex);                /* leave critical region */
        up(&empty);                /* increment count of empty slots */
        consume_item(item);        /* do something with the item */
    }
}
```

Fig: The producer-consumer problem using semaphores.

This solution uses three semaphore.

1. **Full**: For counting the number of slots that are full, initially 0
2. **Empty**: For counting the number of slots that are empty, initially equal to the no. of slots in the buffer.
3. **Mutex**: To make sure that the producer and consumer do not access the buffer at the same time, initially 1.

Semaphore (cont.):

Here in this example semaphores are used in two different ways.

1. **For mutual Exclusion:**
   - The mutex semaphore is for mutual exclusion. It is designed to guarantee that only one process at at time will be reading or writing the buffer and the associated variable.

2. **For synchronization:**
   - The full and empty semaphores are needed to guarantee that certain certain event sequences do or do not occur. In this case, they ensure that producer stops running when the buffer is full and the consumer stops running when it is empty.
   - The above definition of the semaphore suffer the problem of busy wait.
   - To overcome the need for busy waiting, we can modify the definition of the P and V operation of the semaphore.
   - When a Process executes the P operation and finds that the semaphores value is not positive, it must wait.
   - However, rather than busy waiting, the process can block itself. The block operation places into a waiting queue associated with the semaphore, and the state of the process is switched.

Semaphore (cont.):

Advantages of semaphores:

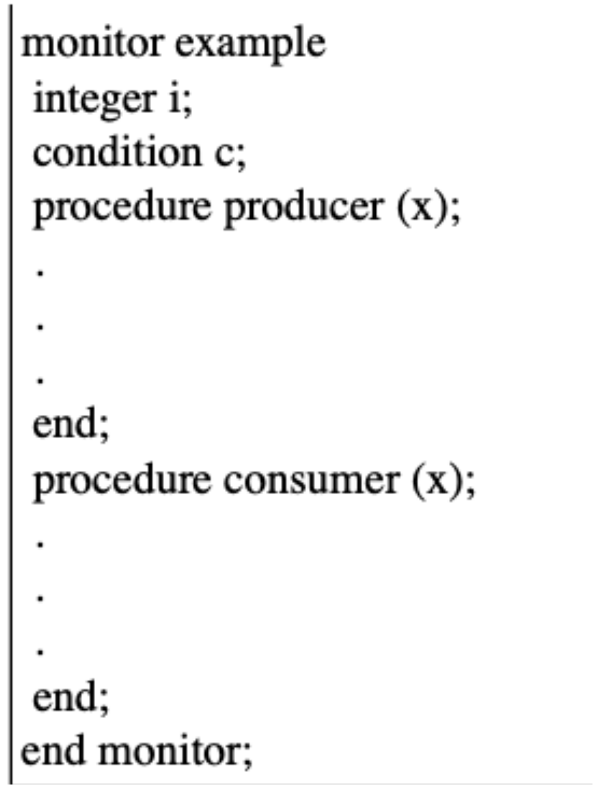
- Processes do not busy wait while waiting for resources. While waiting, they are in a "suspended'' state, allowing the CPU to perform other chores.
- Works on (shared memory) multiprocessor systems.
- User controls synchronization.

Disadvantages of semaphores:

- **can only be invoked by processes-**-not interrupt service routines because interrupt routines cannot block
- **user controls synchronization-**-could mess up.

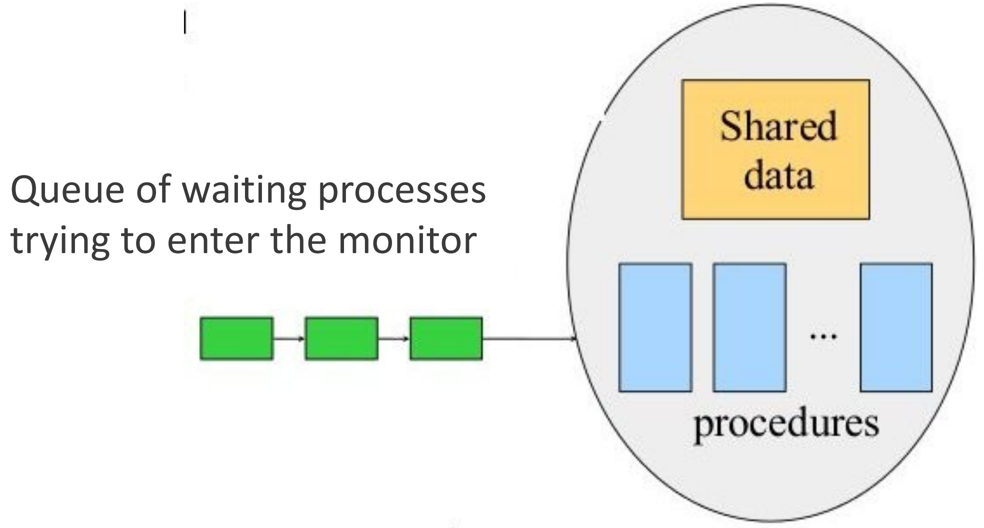
Monitors:

- **In concurrent programming, a monitor is an object or module intended to be used safely by more than one thread.**
- The defining characteristic of a monitor is that its methods are executed with mutual exclusion.
- That is, at each point in time, at most one thread may be executing any of its methods.
- This mutual exclusion greatly simplifies reasoning about the implementation of monitors compared to reasoning about parallel code that updates a data structure.



```
monitor example
 integer i;
 condition c;
 procedure producer (x);
 .
 .
 .
 end;
 procedure consumer (x);
 .
 .
 .
 end;
end monitor;
```

Example of Monitor written in Pascal Language



Monitors (cont.):

- With semaphores IPC seems easy, but Suppose that the two downs in the producer's code were reversed in order, so mutex was decremented before empty instead of after it.
- If the buffer were completely full, the producer would block, with mutex set to 0.
- Consequently, the next time the consumer tried to access the buffer, it would do a down on mutex, now 0, and block too.
- Both processes would stay blocked forever and no more work would ever be done. This unfortunate situation is called a deadlock.

Features

- A higher level synchronization primitive.
- A monitor is a collection of procedures, variables, and data structures that are all grouped together in a special kind of module or package.
- Processes may call the procedures in a monitor whenever they want to, but they cannot directly access the monitor's internal data structures from procedures declared outside the monitor.
- This rule, which is common in modern object-oriented languages such as Java, was relatively unusual for its time,

Message Passing:

- **Message passing in computer science, is a form of communication used in parallel computing, object-oriented programming, and inter-process communication**.
- In this model processes or objects can send and receive messages (comprising zero or more bytes, complex data structures, or even segments of code) to other processes.
- By waiting for messages, processes can also synchronize.

Message passing is a method of communication where messages are sent from a sender to one or more recipients.

Forms of messages include **(remote) method invocation, signals, and data packets**.

When designing a message passing system several choices are made:

- Whether messages are transferred reliably
- Whether messages are guaranteed to be delivered in order
- Whether messages are passed one-to-one, one-to-many (multicasting or broadcasting), or many- to-one (client–server).
- Whether communication is synchronous or asynchronous.

Message Passing (cont.):

- **This method of inter-process communication uses two primitives, send and receive**, which, like semaphores and unlike monitors, are system calls rather than language constructs.
- As such, they can easily be put into library procedures, such as

  send(destination, &message); and

  receive(source, &message);

- **Synchronous** message passing systems requires the sender and receiver to **wait for each other to transfer the message.**
- **Asynchronous** message passing systems deliver a message from sender to receiver, **without waiting for the receiver to be ready.**

Classical IPC Problems

1. Dining Philosophers Problem
2. The Readers and Writers Problem
3. The Sleeping Barber Problem

1. Dining Philosophers Problem

- There are **N philosophers sitting** around a **circular table eating spaghetti** and **discussing philosophy.**
- **The problem is that each philosopher needs 2 forks (chopstick or spoon) to eat, and there are only N forks, one between each 2 philosophers.**
- **Design an algorithm that the philosophers can follow that insures that none starves as long as each philosopher eventually stops eating, and such that the maximum number of philosophers can eat at once.**

The problem was designed to illustrate the problem of avoiding deadlock, a system state in which no progress is possible.

One idea is to instruct each philosopher to behave as follows:

- **think until the left fork is available; when it is, pick it up**
- **think until the right fork is available; when it is, pick it up**
- **eat**
- **put the left fork down**
- **put the right fork down**
- **repeat from the start**

This solution is incorrect: it allows the system to reach deadlock.

1. Dining Philosophers Problem (cont.)

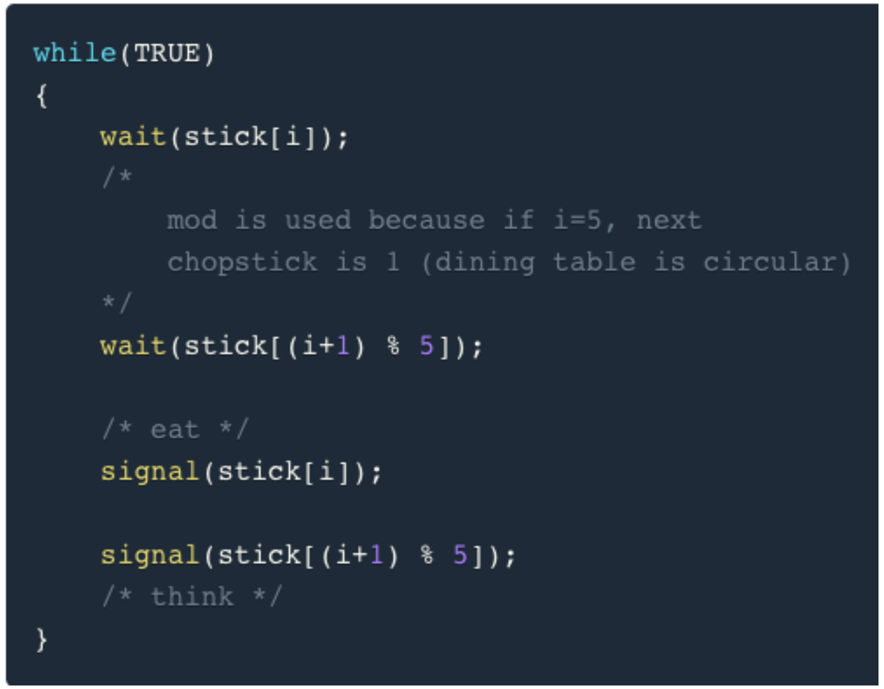
- **Suppose that all five philosophers take their left forks simultaneously. None will be able to take their right forks, and there will be a deadlock.**
- We could modify the program so that after taking the left fork, the program checks to see if the right fork is available.
- If it is not, the philosopher puts down the left one, waits for some time, and then repeats the whole process.
- This proposal too, fails, although for a different reason.
- With a little bit of bad luck, all the philosophers could start the algorithm simultaneously, picking up their left forks, seeing that their right forks were not available, putting down their left forks, waiting, picking up their left forks again simultaneously, and so on, forever.
- **A situation like this, in which all the programs continue to run indefinitely but fail to make any progress is called starvation.**

Solution to Dining Philosophers Problem

- **A solution of the Dining Philosophers Problem is to use a semaphore to represent a chopstick.**
- A chopstick can be picked up by executing a wait operation on the semaphore and released by executing a signal semaphore.
- The structure of the chopstick is shown below –

  semaphore stick [5];

- Initially the elements of the chopstick are initialized to 1 as the chopsticks are on the table and not picked up by a philosopher.



```
while(TRUE)
{
    wait(stick[i]);
    /*
        mod is used because if i=5, next
        chopstick is 1 (dining table is circular)
    */
    wait(stick[(i+1) % 5]);

    /* eat */
    signal(stick[i]);

    signal(stick[(i+1) % 5]);
    /* think */
}
```

- When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick.
- Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down.
- **But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs because they will be waiting for another chopstick forever.**

Difficulty with the solution

- **The above solution makes sure that no two neighboring philosophers can eat at the same time. But this solution can lead to a deadlock.**
- This may happen if all the philosophers pick their left chopstick simultaneously. Then none of them can eat and deadlock occurs.

- **Some of the ways to avoid deadlock are as follows –**
  - There should be at most four philosophers on the table.
  - An even philosopher should pick the right chopstick and then the left chopstick while an odd philosopher should pick the left chopstick and then the right chopstick.
  - A philosopher should only be allowed to pick their chopstick if both are available at the same time.

Readers Writer Problem

- Readers writer problem is another example of a classic synchronization problem.

Problem Statement

- The readers-writers problem relates to an object such as a file that is shared between multiple processes.
- **Some of these processes are readers i.e. they only want to read the data from the object and some of the processes are writers i.e. they want to write into the object.**
- The readers-writers problem is used to manage synchronization so that there are no problems with the object data.
- For example - If two readers access the object at the same time there is no problem.
- **However if two writers or a reader and writer access the object at the same time, there may be problems.**

Solution to Readers Writer Problem

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one **mutex** `m` and a **semaphore** `w`. An integer variable `read_count` is used to maintain the number of readers currently accessing the resource. The variable `read_count` is initialized to `0`. A value of `1` is given initially to `m` and `w`.

Instead of having the process to acquire lock on the shared resource, we use the mutex `m` to make the process to acquire and release lock whenever it is updating the `read_count` variable.

The code for the **writer** process looks like this:

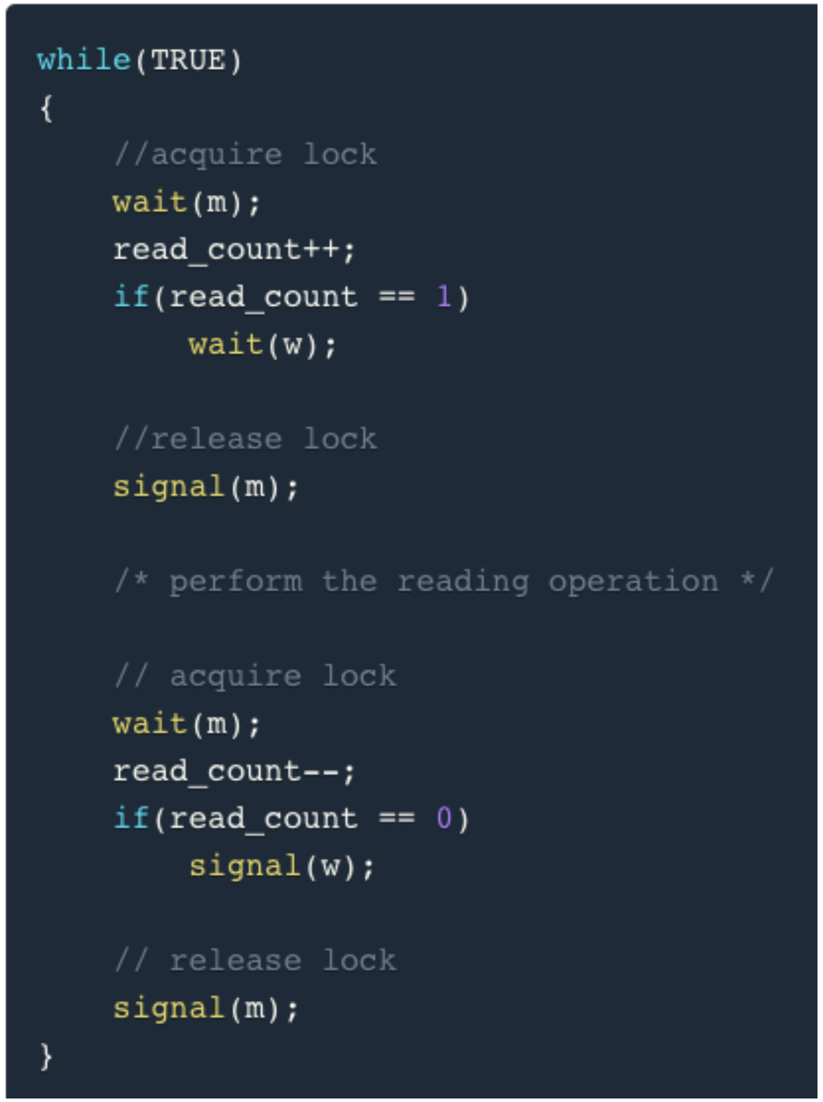
```
while(TRUE)
{
    wait(w);

   /* perform the write operation */

   signal(w);
}
```

Solution to Readers Writer Problem (cont.)

And, the code for the **reader** process looks like this:



```
while(TRUE)
{
    //acquire lock
    wait(m);
    read_count++;
    if(read_count == 1)
        wait(w);

    //release lock
    signal(m);

    /* perform the reading operation */

    // acquire lock
    wait(m);
    read_count--;
    if(read_count == 0)
        signal(w);

    // release lock
    signal(m);
}
```

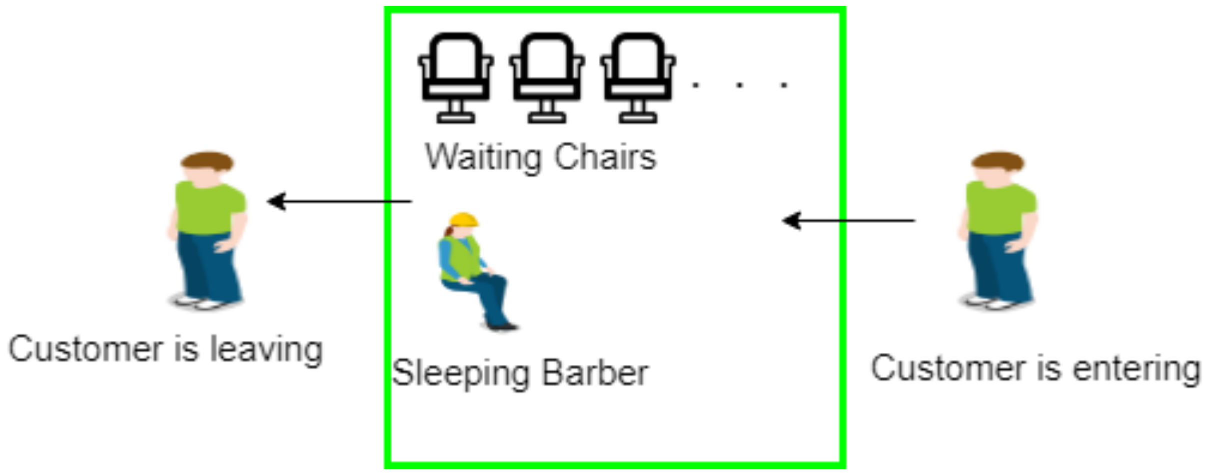
Here is the Code uncoded(explained)

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.
- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

Sleeping Barber Problem

Problem : The analogy is based upon a hypothetical barber shop with one barber. There is a barber shop which has one barber, one barber chair, and n chairs for waiting for customers if there are any to sit on the chair.

- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he has to wake up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.



Solution to Sleeping Barber Problem

- Many possible solutions are available.
- **The key element of each is a mutex**, which ensures that only one of the participants can change state at once.
- The barber must acquire this mutex exclusion before checking for customers and release it when he begins either to sleep or cut hair.
- A customer must acquire it before entering the shop and release it once he is sitting in either a waiting room chair or the barber chair.
- This eliminates both of the problems mentioned in the previous section.
- A number of semaphores are also required to indicate the state of the system. For example, one might store the number of people in the waiting room.
- A multiple sleeping barbers problem has the additional complexity of coordinating several barbers among the waiting customers